"Propagators for table constraints"

Mairy, Jean-Baptiste

Abstract

Constraint Programming is devoted to finding solutions to hard combinatorial problems. Such problems usually define exponentially large search spaces. Coping with such search spaces to find solution(s) (or the best solution) is only possible through the use of sophisticated techniques. Several techniques exist and this thesis is focused on one of them: propagation. Propagation aims at removing parts of the search space that provably contain no solution. The constraints are used to locate (and remove) such parts of the search space. The propagation considered in this thesis is that of the Table Constraint. Table constraints are constraints that give explicit access to the list of allowed tuples. Three chapters cover different aspects of propagation for this constraint. In the first one, five different Generalized Arc Consistency (GAC) propagators are proposed for table constraint. Two of them have an optimal time complexity. All the proposed propagators are evaluated on a variety of b...

Document type : *Thèse (Dissertation)*

PROPAGATORS FOR TABLE CONSTRAINTS

JEAN-BAPTISTE MAIRY

Thèse présentée en vue de l'obtention du grade de docteur en sciences de l'ingénieur

2014

Institute of Information and Communication Technologies,
Electronics and Applied Mathematics
Louvain School of Engineering
Louvain-la-Neuve
Belgium

Thesis Committee:

Yves Deville (director)	Université catholique de Louvain, Belgium
Charles Pecheur	Université catholique de Louvain, Belgium
Christophe Lecoutre	CRIL-CNRS, Université d'Artois, France
Pierre Schaus	Université catholique de Louvain, Belgium
Peter Van Roy (president)	Université catholique de Louvain, Belgium

ABSTRACT

Constraint Programming is devoted to finding solutions to hard combinatorial problems. Such problems usually define exponentially large search spaces. Coping with such search spaces to find solution(s) (or the best solution) is only possible through the use of sophisticated techniques. Several techniques exist and this thesis is focused on one of them: propagation.

Propagation aims at removing parts of the search space that provably contain no solution. The constraints are used to locate (and remove) such parts of the search space. The propagation considered in this thesis is that of the Table Constraint. Table constraints are constraints that give explicit access to the list of allowed tuples. Three chapters cover different aspects of propagation for this constraint. In the first one, five different Generalized Arc Consistency (GAC) propagators are proposed for table constraint. Two of them have an optimal time complexity. All the proposed propagators are evaluated on a variety of benchmarks against the state-of-the-art propagators for table constraints. The experimental results show that our propagators are faster than the state of the art when the arity of the tables is between 3 and 4 (inclusive). For binary table constraints, they are outperformed by propagators dedicated only to binary constraints. When the arity is strictly greater than 4, our propagators are competitive with the state of the art. The second chapter covers both GAC propagation and the expressivity of constraint programming. It presents a generalization of table constraint, called Smart Table Constraint, together with its GAC propagator, called smartSTR2. Smart table constraints introduce simple arithmetic expressions inside the allowed tuples. This improves the expressivity and allows an efficient filtering of this new constraint. After presenting

the syntax, semantics, and the GAC propagator, that chapter will experimentally compare smartSTR2 with the state-of-the-art GAC propagators on several global constraints. Smart table constraint indeed allow an efficient representation of several well known global constraints. The third chapter is dedicated to a consistency stronger than GAC for table constraints, called Domain k-Wise Consistency (DkWC), and a procedure to easily enforce it. This procedure is based on the pre-search computation of a modified CSP such that enforcing GAC on this CSP amounts to enforcing DkWC on the original one. Existing GAC propagators for table constraints can thus be used without any modification to enforce this stronger consistency. Unfortunately, enforcing DkWC is costly. This is also the case for the other consistencies stronger than GAC. We thus also propose two weaker variants of our filtering procedure that are still stronger than GAC but less costly to enforce than full DkWC. These weaker variants, more practical, are compared on a variety of benchmarks to state-of-the-art GAC propagators as well as state-of-the-art propagators for consistencies stronger than GAC.

Throughout this thesis, all the proposed algorithms are evaluated on different benchmarks against other alternatives. The results of these evaluations are measurements made on executions of different programs. Those measurements can be tricky to analyze from a statistical viewpoint because they contain missing data (for instance, an algorithm failing to solve an instance within a given time budget) and hypotheses on their distributions are hard to make. In this thesis, we developed a statistical procedure, based on the bootstrap method, to compare algorithms in this context. This procedure is applied to the experimental results presented in this thesis.

ACKNOWLEDGMENTS

I'd like first to warmly thank Yves Deville, my supervisor, for its guidance during the last 4 years and a half. His constant support, his numerous advices and the many technical discussions we had made the fulfilment of this thesis possible. I also would like to thank Yves for the many discussions, other than technical, that we had during these times. I learnt a lot of things from you !

Going to the department each day was a pleasure, thanks to the people who work there. First of all, I'd like to thank my former and present office mates. With Florence and Vianney, the discussions in the office where always interesting, were they scientific, para-scientific or not at all scientific. I was lucky enough to find the same serious but relaxed atmosphere with Cyrille and Ratheil as office mates. Within the office, we even had our own set of manners! I also would like to thank my friends and colleagues Fanfwè, Xavier, Jey, Guillaume-Bernard, Karim, Nico, Sam, Romain, Simon ant Antoine. Going through a PhD from the beginning to the end with you guys was pleasurable and really supporting. Many thanks also go to all the people working at the INGI department. Weather it is for the seminars, the social events, the coffee breaks, the day to day organization, etc., you guys are the best !

Without the careful reading, intelligent questions and numerous comments of the jury, this thesis would not be what it is. I sincerely thank all the members of the jury for this. Special thanks go to Christophe Lecoutre, with whom the different collaborations have been as fruitful as they have been enjoyable.

On the personal level, carrying out a PhD would not be possible without a supporting and loving family. I thus hereby thank my parents, brothers, my grandfather and my godfather for their support.

During these four years and a half, I have had the chance to have the most wonderful support one can have: the nicest life partner there is. I'd like to give my deepest gratitude to Céline, who supported, comforted and motivated me from day to day during the whole process. You just rock baby !

I gratefully acknowledge the support of the *Fond National de la Recherche Scientifique* (FNRS).

CONTENTS

INTRODUCTION

PROPAGATION IN CONSTRAINT PROGRAMMING

The context in which this thesis lies is Constraint Programming. Constraint programming is a paradigm to solve problems (aren't we all trying to solve problems?). The problems tackled by constraint programming are described in a particular form: as constraint programs. Constraint programs describe what the solution to the problem is, not how to obtain it. This description is made in the form of constraints the solution(s) have to respect. Constraint programs also define a search space. The search space is the space in which to search for a solution to the problem (or the best solution to the problem in terms of some criterion). This description of a problem as a constraint program is called a Constraint Satisfaction Problem (CSP) when the goal is to find solution(s) satisfying the constraints inside the search space, and a Constraint Optimization Problem (COP) when the best solution is searched for. The search space containing all the candidate solutions is usually too large to be searched exhaustively without using sophisticated techniques. Several such techniques exist. One of them will be of interest for this thesis. This technique detects and ignores parts of the search space where there is provably no solution while doing an exhaustive search. This technique, called propagation, has the advantage of being complete: if a solution exists and you have enough time, the technique will find it. This is granted by the property that the removed parts of the search space do not contain any solution.

More practically, CSPs model real world problems with three components: the variables, a domain for each variable, and a set of constraints. Each variable's domain defines the possible values that that variable is allowed to take.

They thus define the search space. Each constraint puts restrictions on the allowed combinations of values for a subset of the variables. This subset is called the scope of the constraint. Solving a CSP means finding solution(s). A solution consists of a value for each variable from its domain such that each constraint is satisfied. COPs have one additional component: an objective function to optimize. The objective function assigns values to the candidate solutions. The goal is then to find the best solution in terms of the objective function.

Propagation is the task of detecting and removing those parts of the search space where there are provably no solution. It uses the constraints on the solutions to deduce and propagate information on the search space while the exhaustive search takes place. The propagation is usually characterized by a consistency. Consistencies are properties of the CSPs or COPs that the propagation enforces. For instance, a well known consistency requires that each value in the domains of the variables participates in at least one solution to each constraint in isolation from the others. Enforcing the consistency amounts to removing the values from the domains of the variables that do not satisfy the property. Reducing the domains of the variables reduces the search space. One can also tighten the constraints to reduce the search space.

The propagation takes place in so-called propagators. Propagators are often associated with constraints since the deduction is based on them, often in isolation from the other constraints. We will be particularly interested in one constraint: the Table Constraint. This constraint simply gives explicit access to the raw list of accepted combinations of values for the variables in its scope. The raw list is called the table. It allows modeling any possible constraint. Propagating a table constraint presents several challenges: the tables of allowed combinations of values are usually long and there is no general structure inside the tables (since they can model any constraint). This means that usually the tables have to be traversed in order to deduce and propagate information. In this thesis, several new propagators are introduced for table constraints. Five of them are associated with the table constraints individually and one of them is a procedure allowing of integrating the propagation information of several table constraints to prune the search space more. We also define the consistency associated with the last propagator: the Domain k-Wise Consistency. A new constraint, generalizing the table constraint, is also introduced together with its propagator. This new constraint is called the Smart Table Constraint. It generalizes the expression of tables by introducing simple arithmetic expressions that can be used inside the tables of the constraints.

When designing a new propagator or propagation algorithm, it is common to compare it with existing state-of-the-art alternatives. The results of such comparisons are measurements of the execution of the algorithms. Being able

to draw conclusions based on those results is not always obvious. Some algorithms may even fail to solve some problem instances within the given resource budget, leading to incomplete data. In order to deal with such results, we have developed a statistical procedure. This procedure can be used to draw conclusions on the relative performance of techniques or algorithms. It is applied to all the experimental data presented in this thesis.

CONTRIBUTIONS

The contributions of this thesis are:

- The introduction of five new propagators, two of them having an optimal time complexity, for the table constraint;
- The experimental evaluation of those propagators against state of the art propagators for table constraints;
- The introduction of the Smart Table Constraint;
- The definition of an efficient propagator for the smart table constraint and the experimental evaluation of this propagator;
- The definition of Domain k-Wise Consistency;
- The introduction of a procedure, relying only on existing propagators, to obtain domain k-wise consistency on table constraints and its experimental evaluation;
- The definition of a procedure, relying on a statistical background, to analyse experimental data;
- The application of this statistical procedure to all the experimental data of this thesis.

  All the code developed during this thesis has been published under the GPLv3 license and can be found on `http://becool.info.ucl.ac.be/resources/table-constraint-propagators`.

OUTLINE

This thesis is organized into two parts. The first part gives the background and state-of-the-art information about all the contents of this thesis. The second part is concerned with the propagation in constraint programming applied to the table constraint.

In Part I, Chapter 1 gives background information on Constraint Programming with a particular focus on one constraint heavily studied in this thesis: the Table Constraint. Chapter 2 presents propagation in constraint programming, again with a focus on table constraints. Finally, Chapter 3 defines the statistical procedure used throughout this thesis to further analyse the experimental results of the algorithms.

Part II is divided into three chapters. The first one, Chapter 4, presents five different propagators for the table constraints, amongst which, two have an optimal time complexity. Chapter 5 defines a new generalized form of table constraint, called smart table constraint, together with a generalized arc consistency propagator for this new constraint. Chapter 6 proposes a filtering procedure to obtain stronger filtering than generalized arc consistency on table constraints, relying only on existing propagators. All the chapters present the corresponding experimental results.

PUBLICATIONS

Journal Publications

[DVHM13] Yves Deville, Pascal Van Hentenryck and Jean-Baptiste Mairy, ”Domain Consistency with Forbidden Values”, Constraints 18 (3), pages 377-403

[MVHD14a] Jean-Baptiste Mairy, Pascal Van Hentenryck and Yves Deville, "Optimal and Efficient Filtering Algorithms for Table Constraints", Constraints 19 (1), pages 77-120

Conference Publications

[MVHD12] Jean-Baptiste Mairy, Pascal Van Hentenryck and Yves Deville, "An Optimal Filtering Algorithm for Table Constraints", 18th International Conference on Principles and Practice of Constraint Programming (CP 2012), 2012, pages 496–511.

[MDL14] Jean-Baptiste Mairy, Yves Deville and Christophe Lecoutre, "Domain k-Wise Consistency Made as Simple as Generalized Arc Consistency", Integration of AI and OR Techniques in Constraint Programming (CPAIOR) 2014, pages 235-250

[MDL15] Jean-Baptiste Mairy, Yves Deville and Christophe Lecoutre, "The Smart Table Constraint", Integration of AI and OR Techniques in Constraint Programming (CPAIOR) 2015

Workshop publications

[MSD10] J.B Mairy, P. Schaus and Y. Deville, “Generic Adaptive Heuristics for Large Neighborhood Search”, Workshop on Local Search Techniques in Constraint Satisfaction of the 16th International Conference on Principles and Practice of Constraint Programming (CP 2010), 2010.

[MDVH11] J.B Mairy, Y. Deville, and P. Van Hentenryck, "Reinforced Adaptive Large Neighborhood Search", Workshop on Local Search Techniques in Constraint Satisfaction of the 17th International Conference on Principles and Practice of Constraint Programming (CP 2011), 2011.

Part I

BACKGROUND

1

CONSTRAINT PROGRAMMING

Constraint Programming (CP) is a paradigm to solve hard combinatorial problems [RBW06, Lec09]. Combinatorial problems are problems for which a solution (or the best solution) is searched for inside a search space. It is the paradigm governing all the work in this thesis. The search space can be seen as the space of possible candidate solutions. The search spaces of combinatorial problems are, generally, too large to be searched exhaustively. Many of such problems belong to the class of NP-hard problems. In constraint programming, the problems are described in a declarative fashion, meaning that the search space is described together with the constraints on the solution. In other words, the problem describes what the solutions are like, not how to obtain them. The goal is then to find a candidate solution inside the search space that respects all the constraints, or the best candidate solution that respects all the constraints. Those problems for which only a solution is sought are called Constraint Satisfaction Problems (CSPs). Problems for which the best solution is sought are called Constraint Optimization Problems (COPs). Section 1.1 presents the CSP framework, Section 1.2 presents the COP framework, and Section 1.3 presents one particular constraint that is studied in this thesis.

					3		8	5
		1		2				
			5		7			
		4				1		
	9							
5							7	3
		2		1				
				4				9

Figure 1.1: Example of a Sudoku

1.1 CONSTRAINT SATISFACTION PROBLEMS

Constraint satisfaction problems are described by a set of decision variables, a domain (the possible values) for each of the variables, and a set of constraints on the variables. The goal is to find values for each decision variable, inside its domain, such that all the constraints are satisfied. The domains of the variables can be continuous or discrete. In this thesis, we will only focus on discrete domains for the variables. Example 1 gives the CSP encoding for the well known Sudoku problem.

Example 1. The Sudoku problem is the problem of filling a 9×9 grid with digits from 1 to 9. Some positions of the grid are pre-filled with digits. All the digits in a row or a column must be different from each other. Also, each of the 3-by-3 sub-grids must contain all the digits from 1 to 9. An example of a sudoku problem can be found in Figure 1.1. The Sudoku problem can be described as a CSP. One possible CSP model for that problem is to have one variable per position in the grid, for a total of 81 decision variables. Each variable has $\{1,\ldots,9\}$ as its domain. There is one constraint per row and one constraint per column, constraining the variables of the row/column to take different values. There is also one constraint per 3×3 sub-grid, constraining the variables of the sub-grid to take different values. Finally, there is one constraint per pre-filled position in the Sudoku, constraining that variable to take the pre-defined value.

In Example 1, we can already see that the size of the search space is huge. Indeed, each of the 81 variables can take 9 different values. The size of the search space is thus 9^{81}. Even with 0.01 ms to check that the solutions respect the constraints (pre-defined positions, digits in rows and columns different, and digits in the 3×3 sub-grids different), checking all the possible candidate solutions would take around 6×10^{58} million years. But you can solve this CSP in less time than that, can't you? So does the constraint programming framework!

Formally, CSP (X, D, C) is composed of an ordered set of *n variables* $X = \{x_1, \ldots, x_n\}$, a set of *domains* $D = \{D(x_1), \ldots, D(x_n)\}$ where $D(x_i)$ is the set of possible *values* of the variable x_i, and a set of *e constraints* $C = \{c_1, \ldots, c_e\}$, where each constraint c_j restricts the possible combinations of values, called *allowed tuples*, for a subset of variables from X: this subset is called the *scope* of c_j and is denoted by $scope(c_j)$. The arity of a constraint c_j is the size of its scope: $\#scope(c_j)$. Because variable domains may evolve (be reduced) during the search, $D(x)$ is referred to as the *current domain* of x, which is a subset of the *initial domain* of x (denoted by $D^{init}(x)$). For a group of variables $Y \subseteq X$, $D(Y)$ denotes the set of domains for the variables in Y. We will use the term *literal* to refer to a variable–value pair. An assignment of a set of variables $Y = \{y_1, \ldots, y_k\}$ is a set of literals $\{(y_1, v_1), \ldots, (y_k, v_k)\}$ with $(v_1, \ldots, v_k) \in D^{init}(Y)$; it is a *valid* assignment if $(v_1, \ldots, v_k) \in D(Y)$. An assignment of the set of variables Y is *total* for a CSP (X, D, C) if $Y = X$ (i.e., it assigns a value to every variable). An assignment or tuple τ *satisfies* a set of constraints C' iff for every constraint $c' \in C'$, $\tau[scope(c')]$ is allowed by c', where $\tau[Y]$ denotes the restriction of τ to literals referring to variables in Y. The goal of the search is to find a *valid total assignment* that respects all the constraints. The set of solutions of a CSP $P = (X, D, C)$ is denoted by $sols(P)$.

Each CSP can be associated with a *constraint graph*. This (hyper-)graph has one vertex per variable and one (hyper-)edge per constraint, connecting the vertices representing the variables of the constraint scope. This graph is also called the *primal* constraint graph. Another graph representing a CSP is the *dual* constraint graph. This graph has one vertex per constraint and two constraints are linked iff their scopes share common variables.

Constraint programming is often described by the following equation:

$$CP = Model + Search$$

The *Model* part of the equation is the process by which the real world problem is translated into a triplet (X, D, C). The expressiveness of CP comes from

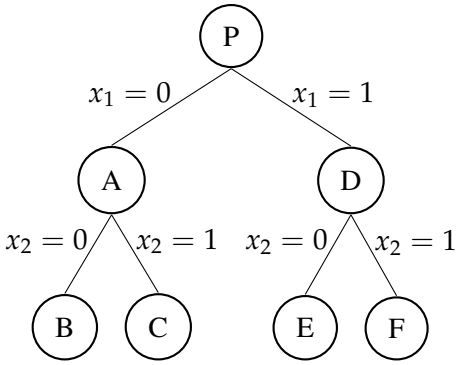


Figure 1.2: Example of a search tree for a CSP

the extensive catalog of constraints available to the user. They range from very simple arithmetic constraints to complex constraints such as the geost constraints. This constraint states that the shapes defined by shifted boxes represented with its variables do not overlap in the set of specified dimensions. Adequately choosing the decision variables, their domains, and the constraints, has a large impact on the performance of a constraint programming program for solving a problem. For instance, in Example 1, setting the domain of the variables representing the pre-filled position to the value in the sudoku reduces the search space to 9^{64}, since those variables can only take one value. This divides the size of the search space by 1.6×10^{16}.

The *Search* part of the equation relates to the way a solution is sought inside the search space. The search space is concretized as a tree. Each node of the tree is a CSP, with the root of the tree being the original CSP. Arcs of the tree represent decisions. A decision is usually a modification of the domain of a variable. For instance, it could be the assignation of a value to a variable or a refutation of a value from a domain. Example 2 shows an example of a search tree where the decisions are assignments.

Example 2. Figure 1.2 gives an example of a search tree. In this example, the CSP to solve is a CSP P with two binary variables x_1 and x_2 and some constraints C. We thus have $P = (X, D, C)$ with $X = \{x_1, x_2\}$ and $D = \{D(x_1), D(x_2)\}$ where $D(x_1) = D(x_2) = \{0, 1\}$. In this example, A, B, C, D, E and F are CSPs, each of which is obtained by taking the decision that is on the edge to the parent CSP. Those decisions can be expressed by modifying the domains of the variables or adding the constraint to the child CSP. The leaves of the search tree contain all the full valid assignations. The goal is to find one that respects the constraints.

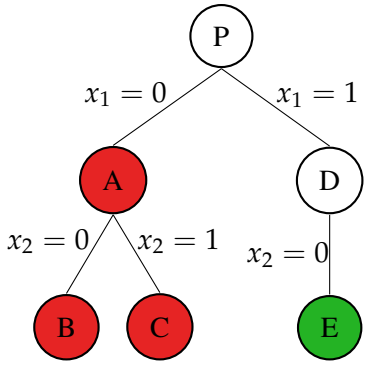


Figure 1.3: Exploration of a search tree. Nodes in red are nodes that are failures and the node in green is a solution to the CSP.

The search tree is constructed on the fly, while searching for the solution(s). This is the concept of *backtracking search*. A backtracking search builds the search tree while exploring it. When an inconsistency is detected at a node (such as the violation of a constraint), the search backtracks to a previous search node and makes a different choice (leading to a different search node). Example 3 shows the exploration of a search tree.

Example 3. Figure 1.3 shows a possible exploration of the search tree from the CSP of Example 2, where the constraint set only contains the constraint $x_1 - 1 = x_2$. In this example, the tree is traversed with a depth first search traversal. First, the decision $x_1 = 0$ is inspected. From this node, $x_2 = 0$ is tested. The CSP B is a failure because it contains a full valid assignment of the variables that does not respect the constraint. Hence, the search backtracks to A to take the second decision for x_2: $x_2 = 1$. This is also a failure for the same reasons as C. The search backtracks again to A. This time, no further possibility is left for x_2. A is then also a failure. The search then backtracks from A to P to take the other decision for x_1. Finally, the search arrives at E, which is a solution to P. As the search tree is constructed on the fly, node F from Figure 1.2 is never created.

When backtracking to a previous search node, some data structures may have to be restored. Such data structures, maintained during the search, are said to be *trailed* structures.

Another important concept of the search in constraint programming is the *propagation*. Propagation in CP uses the constraints to reduce the search space without removing any solutions. This is done by only ignoring those parts of the search space where there is provably no solution. Propagation can be done

at the beginning of the search (on the original CSP), on some nodes of the search tree, or on all nodes of the search tree. Example 4 illustrates propagation for the case of a simple CSP. Propagation is the topic of Chapter 2.

Example 4. Let's have a look at the CSP used in Example 2. We have $P = (X, D, C)$ with $X = \{x_1, x_2\}$ and $D = \{D(x_1), D(x_2)\}$ where $D(x_1) = D(x_2) = \{0, 1\}$ and $C = \{x_1 - 1 = x_2\}$. Without searching, we can safely remove 1 from the domain of x_2 because no value from the domain of x_1 can satisfy the constraint when $x_2 = 1$. Also, we can remove 0 from the domain of x_1 because no value in the domain of x_2 can satisfy the constraint. Those reductions of the search space are safe because, as there is no way to satisfy the constraint $x_1 - 1 = x_2$ with the values of the initial domains of the variables, those values can never lead to a solution. In this case, the propagation alone reduces the domains of the variables to singletons, meaning that a solution is found.

The parts of the search space that are not considered during the search are the ones that provably contain no solutions. Thus, if a solution exists, it will be found. This is why Constraint Programming is classified among the *complete* search techniques.

1.2 CONSTRAINT OPTIMIZATION PROBLEMS

Constraint Optimization Problems (COPs) are CSPs with the addition of an objective function to optimize. They are described as a quadruplet (X,D,C,O) where X is the set of variables, D is the set of domains for the variables, C is the set of constraints, and O is the objective function. An example of a COP is given in Example 5.

Example 5. The well known Travelling Salesman Problem (TSP) can be seen as a COP. The objective of this problem is, given a list of n cities with the distances between them, to find a tour visiting all the cities only once with the shortest possible total distance. The problem can be formulated as a quadruplet (X, D, C, O). The variables $X = \{x_1, x_2, \ldots, x_n\}$ represent the tour, with x_i containing the identifier of the i^{th} city to visit. The domains of the variables are the possible cities. The constraints C constrain the variables to represent a valid tour for a TSP, i.e., a tour where all the cities are visited only once. The objective function to minimize is the sum of the distances between all the consecutive cities in the tour.

The resolution of a COP in the CP framework is somewhat similar to that of a CSP. It uses the same kinds of backtracking searches and propagation techniques. The difference is that, since the best solution is sought, the search

cannot stop as soon as a solution is found. All the solutions must be inspected in order to find the best one. The objective of the COP can be used to further prune the search space, by constraining, each time a solution is found, the objective value to be better than the last found solution.

1.3 TABLE CONSTRAINTS

Among the extensive catalog of constraints available to the user, global constraints play a key role in the success of CP. Global constraints are constraints that can be used on any number of variables. An example of a global constraint is the *AllDifferent* constraint, stating that all the variables given as arguments take different values. The advantage of using one global constraint, as opposed to all the binary "$\neq$" constraints, is that the global constraint captures the semantics of all the binary constraints at once. It allows their propagation to perform stronger search space reductions and/or to do it more efficiently.

AllDifferent is an example of a constraint given in intention: the semantics of the constraint is given, but the allowed combinations of values for the variables are not listed explicitly. In contrast, *table constraints* are global constraints that explicitly list the set of allowed combinations of values for the variables in their scope. This set is called the *table*. A particular combination of values for the variables will be accepted by the constraint iff it is listed in its table. Table constraints are the target constraints of Chapters 4, 5 and 6. Note that table constraints can also list the set of disallowed combinations, but they are not the focus of this thesis.

Table constraints are essential to constraint programming and will be needed more and more. Indeed, they theoretically allow encoding any constraint. They thus allow encoding any constraint for which no intensional form exists. Such constraints frequently arise in real world application, where the data is messy and not everything can be put in equations. Table constraints are, for instance, mandatory when one or more items respecting some constraints are sought inside a set of items, and each item is described by a set of numerical characteristics. In this case, one item is a line in the table. In order for a set of variables to represent the characteristics of one item, a table constraint is needed, because in most cases, there is no arithmetic relation between the characteristics of the items. An example of such an application is the querying of the semantic web. Constraint programming has been used in [dSMDS11, dSMDSC12] to query the semantic web. The goal of the semantic web is to make the information on the internet, which is human readable, machine-readable and to allow efficient semantic querying. In the semantic web, the data contained in a website can be represented as a Resource Description Format (RDF) graph. This graph con-

x	y	z
1	1	2
1	2	3
2	1	3

Figure 1.4: Table constraint encoding for $x + y = z$ where $D(x) = D(y) = \{1,2\}$ and $D(z) = \{2,3\}$

tains nodes, representing resources (e.g.: people, objects, concepts, etc.) and literals, representing values (strings, dates, numbers, etc.). Nodes are linked together by labelled edges, which encode a relation between the two resources. It is possible to represent this graph with a large table constraint of arity 3. Each line represents an edge: the first column contains the origin node, the second, the edge, and the last, the destination node. Queries of the semantic web can also be viewed as (small) graphs. Indeed, the query of the relation between Bob and Alice can be encoded with a simple graph containing two nodes, Bob and Alice, and a variable as the label of the edge. Then this triplet is constrained to appear in the table representing the graph. Of course, much more complicated queries exist and correspond to more complicated graphs. We can thus encode queries in the model of the CSP and the RDF graph as a table constraint. The Semantic Web is an example of the use of CP in the area of big data, and in this example, table constraints are mandatory. More generally, table constraints are also very useful in database applications of constraint programming. Indeed, it is very often only possible to describe tables in a database by table constraints. Table constraints will be more and more needed because constraint programming is being applied to more and more areas where the data is often only representable with table constraints.

To give a simple example, Figure 1.4 gives a table constraint encoding the constraint $x + y = z$ where $D(x) = D(y) = \{1,2\}$ and $D(z) = \{2,3\}$.

Formally, inside a CSP (X, D, C), the table of a table constraint c with $scope(c) = \{x_1, x_2, \ldots, x_r\}$ is a set of tuples on $scope(c)$. Since there is no ambiguity about $scope(c)$, we will write the tuples of the tables omitting the variables. Not all the tuples in a table are required to be valid, as the domains may evolve while the table can stay the same. Given a set of tuples T of arity r, a table constraint c over T holds if $(x_1, \ldots, x_r) \in T$, where $scope(c) = (x_1, \ldots, x_r)$. The size t of a table constraint c is its number of

tuples, which is also denoted by $c.length$. Given a table constraint, we say that a tuple τ is *allowed* if it belongs to the table.

2

PROPAGATION IN CONSTRAINT PROGRAMMING

As seen in Chapter 1, propagation is the mechanism used by constraint programming to reduce the search space without removing any solution. This reduction mechanism uses the constraints to perform deductions and hence reduce the search space. Usually, propagation is characterized by *consistencies*. Consistencies are properties of the CSPs that the propagators enforce. Enforcing a consistency amounts to reducing the search space until the property is respected, without removing any solutions. Consistencies have generated a tremendous research effort. There thus exist many different consistencies and many different ways to enforce them (even for a given consistency). This chapter focuses on the central consistency in constraint programming (Section 2.1), on its application to table constraints (Section 2.2), and on consistencies that prune the search space more than the central consistency (Section 2.3).

2.1 GENERALIZED ARC CONSISTENCY

The central consistency in CP is called Generalized Arc Consistency (GAC), or sometimes Domain Consistency (DC) [Mac77a]. It is the highest filtering

one can obtain by considering the constraints one at a time. It states that all the literals of the variables in the scope of each constraint must participate in at least one solution of the constraint (in isolation from the others). To formally define it, we need first to introduce the notion of support. In the following, for a constraint c, we will denote by $c(\mathbf{v})$ the test evaluating to true iff $\mathbf{v}$ is allowed by c. A literal of a constraint c is a literal (x,a) such that $x \in scope(c)$. A *support* on a constraint c is a tuple $\tau \in D(scope(c))$ such that $c(\tau)$, and a support (on c) for a literal (x,a) of c is a support τ on c such that $\tau[x] = a$. Note that supports are *valid* tuples, meaning that the values involved are necessarily present in the current domains.

Definition 1. (GAC) *A constraint c is* generalized arc consistent *(GAC) iff there exists at least one support for each literal of c. A CSP P is GAC iff every constraint of P is GAC.*

Enforcing GAC is the task of removing from the domains all the values that have no support on a constraint. The values to remove from the domains depend on the constraint. GAC thus reduces the domains of the variables, but this is not the case for all consistencies. Most of the consistencies can be classified in two categories: domain-filtering consistencies and constraint-filtering ones. Domain-filtering consistencies identify inconsistent values that can be removed from the domains of the variables, whereas constraint-filtering ones identify inconsistent tuples in the constraints. GAC is a domain-filtering consistency. Different examples of such consistencies can be found in [Bes06, DB01, BSW08, KWR+10, Lec09, Ste08]. Consistencies can be also ordered with respect to their pruning strength. A consistency stronger than another will reduce the search space more. In contrast, a weaker consistency reduces the search space less. Consistencies weaker than GAC are cheaper to enforce but they lose ground progressively, at least for binary and table constraints, as they reduce (much) less the search space. On the other hand, consistencies stronger than GAC are more and more studied, and often tested on difficult problem instances, where the cost of enforcing them can be counterbalanced by their large inference capabilities. However, such strong consistencies need to reason with several constraints simultaneously, which makes the development of filtering algorithms complex, especially for integration into existing CP solvers. Chapter 6 is concerned with a consistency stronger than GAC.

Alongside the consistencies are the propagation algorithms, enforcing the consistency. A same consistency can be enforced by many different algorithms. Usually, propagation is obtained through a *fixed-point algorithm*. Propagation is performed on each constraint (or group of constraints), which can hopefully reduce the search space. After one constraint (or group of constraint) has performed its propagation, the consistency of the other constraints impacted by the

search space reduction has to be checked again, because the search space has changed and those constraints could prune the search space further. The fixed-point algorithm thus uses a *propagation queue*, containing the constraints that need to (re-)check their consistency. When asking to a constraint (or group of constraints) to (re-)check its consistency, two approaches are possible. The first one just gives to the constraint(s) the information that their consistency has to be checked again. The propagation queue only contains the constraints in this case. The propagators of this kind are called *constraint-based*. The second approach gives to the constraint the literal(s) that have been removed from the domain(s). The propagation queue thus contains constraints together with the literal(s) removed in this case. Propagators using this approach are called *value-based*. Value-based propagators can be more efficient at enforcing the consistency of the constraint(s) but usually have to be called for each literal removed. The propagation algorithm for a consistency can be designed either to be general purpose (working for all constraints) or specialized for one constraint.

For GAC, many general purpose algorithms have been designed, which only need the constraint to be able to check that a particular tuple is allowed. This is the case, for instance, of GAC1 [Mac77a], GAC3 [Mac77b, McG79, CJ98], GAC2001/3.1 [BR01, BRYZ05, ZY01], AC4 [MH86, MM88] AC5 [VDT92], AC6 [Bes94] and AC7 [BFR99]. On the one hand, GAC1, GAC3, GAC2001 and GAC3.1 are constraint based: the propagation queue of their fixed-point algorithm contains only constraints. On the other hand, AC4, AC5, AC6 and AC7 are value-based: the propagation queue of their fixed-point algorithm contains constraints and tuples that have been deleted. Those algorithms are unable to take advantage of the semantics of the constraint, as they can be used to propagate any constraint (general purpose propagators). Many more constraint-based and value-based GAC propagators have been developed for particular constraints. The propagators designed for one constraint use their knowledge of the constraint to be more efficient. Those propagators often use complex algorithms. For instance, the propagator for the AllDifferent constraint [Rég94] maintains a maximal matching between the variables and the values and uses the information of the inclusions of edges into the maximal matching to prune the domains. Making a list of GAC propagators for particular constraints is out of the scope of this thesis. However, regarding GAC, we will be interested in the propagators developed for table constraints. This is the topic of the next section.

2.2 GAC FOR TABLE CONSTRAINTS

A lot of research effort has been spent on GAC propagators for table constraints. To achieve domain consistency, one must at least check the validity of each tuple and, in the worst case, remove all the values from the domains. For a table constraint of arity r containing t tuples with a maximum of d values in the domains, a GAC algorithm has a complexity $\Omega(r \cdot t + r \cdot d)$ per table constraint in the worst case. A classical algorithm used in a fixed-point algorithm with a complexity $O(r \cdot t + r \cdot d)$ per table constraint is thus optimal. Few propagators designed for table constraints have the optimal theoretical complexity, yet, they behave well in practice.

The existing propagators can be categorized in 4 classes: index-based, compression-based, based on a dynamic table, and value-based. Those classes are not mutually exclusive. Each approach is designed to cope, in its own way, with large structure-less tables. The index-based methods build indices of the tuples in the tables so as to traverse them quicker. Compression-based methods build alternative representations of the tables, compressing the information. Dynamic table approaches modify the table during the search to focus only on the relevant parts. Value-based propagators use the value-based propagation paradigm to perform the propagation. As a first point of comparison for the existing techniques, we will analyze the complexity of all the calls to the propagator along a branch of the search tree. We will consider a table constraint of arity r containing t tuples with a maximum of d values in the domains. In this complexity analysis, the time complexity of the backtracking operations is not taken into account. The index-based approaches use an indexing of the table so as to traverse it more quickly. Examples of such propagators are GAC3-allowed and other constraint-based variants (GAC3$_{rm}$-allowed, GAC2001-allowed) [Lec09, BR97, LS06, GJM06]. For each variable–value pair (x, a), the index data structure has an array of the indices of the tuples with value a for x. The space complexity of the data structure is $O(r \cdot t)$. The time complexity of GAC3-allowed along a branch in the search tree is $O(r^3 \cdot d \cdot t)$ per table constraint. Indeed, for one call to GAC3-allowed, a variable requires at worst to test the validity of each tuple of the constraint (thanks to the indexing structure), which costs $O(r \cdot t)$. Thus, one call of GAC3-allowed costs $O(r^2 \cdot t)$. The propagator being called at most $r \cdot d$ times along a branch in the search tree, the complexity of GAC3-allowed is $O(r^3 \cdot d \cdot t)$. GAC2001-allowed has a time complexity of $O(r^2 \cdot t + r^3 \cdot d^2)$ per table constraint along a branch in the search tree. The last support structure of GAC2001-allowed ensures that a tuple from the table is never considered twice, except while checking if the last support is still valid. Along a branch in the search

tree, the cost of testing each tuple once per variable is $O(r^2 \cdot t)$. The cost of the validity tests of the last supports is $O(r^3 \cdot d^2)$, leading to a total complexity of $O(r^2 \cdot t + r^3 \cdot d^2)$. Indexing can also be used in value-based propagators. In [LR05], the authors propose a value-based propagator for table constraints implementing GAC6. It uses a structure which indexes, for each variable–value pair (x, a) and each tuple, the next tuple in the table with value a for x. The spatial complexity of the data structure is $O(r \cdot d \cdot t)$. This space usage can be reduced by using a data structure called a hologram [Lho04]. Another index type, proposed in [GJMN07], indexes, for each tuple and variable, the next tuple having a different value for the variable. Compression-based propagators compress the table in a form that allows a fast traversal. One of such compressed forms uses a trie for each variable [GJMN07]. Another example of a compression-based technique [CY10, Car06] uses a *Multi Valued Decision Diagram* (MDD) to represent the table more efficiently. Their algorithm is called MDD^c. During propagation, the tries or MDD are traversed using the current domains to perform the pruning. These algorithms are constraint-based and have a time complexity of $O(r^2 \cdot d \cdot t)$ per table constraint along a branch in the search tree. This is a worst time complexity, corresponding to the case where there is little or no compression obtained with their respective encodings of the table. In [PR14], the authors propose two new versions of the MDD-based propagator: MDD-4 and MDD-4R. The first version differs from MDD^c by modifying and maintaining the MDD representing the constraint. In MDD^c, the MDD of the constraint is never modified. MDD-4R improves MDD-4 by adding a mechanism to reset the MDD when the cost of the maintenance operations is more expensive than the reset. Compression and faster traversal can also be achieved by using compressed tuples that represent a set of tuples [KW07, XY13, Rég11]. Compressed tuples provide the possibility of using sets instead of values for the variables inside the tuples. The set of tuples represented by a compressed tuple is the Cartesian product of the variables' sets. Another compression method [JN13] is to apply the so-called short supports from [NGJM13] to table constraints. Short supports applied to table constraints allow a tuple to leave variables out, meaning that all the values for that variable are allowed. This allows reducing the size of the tables while proposing an efficient filtering algorithm. Another compression approach is based on the detection of frequent patterns [GHLR14]. Frequent patterns are used to compress tables, and an algorithm is proposed to filter such reduced constraints. Propagators based on dynamic tables maintain the table by suppressing invalid tuples from it. The *or-tools* propagator [PF] maintains such a dynamic table. It uses a bitset on the tuples of the table to maintain their validity. One bitset per literal (x, a) is also used for easy access of the tuples with value a for

variable x. This propagator has an $O(r \cdot d \cdot t)$ time complexity per table constraint along a branch in the search tree. The STR algorithm [Ull07] and its refined versions, STR2, STR2+ [Lec11] and STR2w [LLY14], also maintain a dynamic table. They are constraint-based and scan only the previously valid tuples to extract the domain consistent/inconsistent values. The time complexity of STR2, STR2+ and STR2w is $O(r^2 \cdot d \cdot t)$ per table constraint along a branch in the search tree. This complexity is obtained by multiplying by $r \cdot d$ the complexity of one call to the propagator given in [Lec11, LLY14] while taking into account that the values of the domains can only be removed once along a branch in the search tree. The maximum number of calls to the propagator along one branch in the search tree is indeed $r \cdot d$. None of the previously presented propagators has the optimal $O(r \cdot t + r \cdot d)$ time complexity per table constraint. STR3 [LLY12] has recently been introduced with an optimal time complexity per table constraint. Although the name might suggest it, STR3 is not an improvement of STR2. STR3 is a brand new GAC algorithm for table constraints. It is value-based (while STR2 and STR2+ are constraint-based), thus belonging to the last table constraint propagators category. STR3 starts by precomputing the set of initial supporting tuples for each literal (working with the indexes of those tuples). Those sets are not trailed during the search. Each valid literal has two special supports in its set: the last known valid tuple (called *curr*) and one valid support in this set (we call it *watched*). The tuple *curr* is maintained during the search upon backtracking. Its property is that all the tuples after *curr* are known to be invalid. The second special support is not backtracked during the search. This means that the two can be different. Upon the removal of a literal (x, a), a new valid tuple is searched for all the other literals having (x, a) in their *watched* tuple. This search starts at *curr* towards the head of the set. Once a new valid support is found, both *curr* and *watched* are updated for the literal being inspected. If none is found, then the literal is removed. STR3 can be seen as a highly optimized version of GAC4, applying an idea similar to watched literals to the supports. A recent version of the MDD^c propagator, presented in [GSS11], is both in the value based and compression based categories. This propagator uses an MDD, as does MDD^c, but it never revisits parts of the MDD that do not need to be revisited. To achieve this, it switches from constraint based to value based. It also adds explanations to the MDD propagation, in an incremental fashion.

Table 2.1 gives an overview of the different GAC propagators for table constraints. All these propagators have different theoretical time complexities. They also behave differently in practice. Some are faster than others on some benchmarks but there is no general complete ordering of them with respect to practical speed. What we can see in this table is that only one of the propa-

	Algorithm	Optimal?
Index-based	GAC3-allowed	X
	GAC3$_{rm}$-allowed	X
	GAC2001-allowed	X
Compression-based	trie-based	X
	MDD^c	X
	MDD-4	X
	MDD-4R	X
	compressed tuples	X
	short supports	X
	frequent patterns	X
Dynamic table	or-tool propagator	X
	STR	X
	STR2 / STR2+ / STR2w	X
Value-based	GAC6	X
	STR3	V

Table 2.1: Classification of the existing GAC Propagators for Table Constraint

gator has the optimal time complexity for a branch in the search tree. Chapter 4 presents several new filtering procedures to obtain GAC on table constraints developed in the context of this thesis. Two of them have the optimal $O(r \cdot t + r \cdot d)$ time complexity. The others, although not optimal, perform well in practice.

2.3 CONSISTENCIES STRONGER THAN GAC

Generalized Arc Consistency is the central consistency in constraint programming. It is the strongest filtering achievable while considering constraints in isolation. However, several consistencies stronger than GAC exist. They require considering several constraints at a time but they can prune the search space more than GAC. Consistencies can also be incomparable. Incomparable

consistencies are neither stronger nor weaker than each other: some CSPs can verify one consistency but not the other, and reciprocally. This means that both consistencies can be enforced together to strengthen the pruning. The study of all the consistencies and their relative strengths is outside the scope of this thesis. We will be interested, in this section, in the k-wise consistency and its combinations with GAC.

k-Wise Consistency (kWC) [Jég91, Bes06] can be classified as a constraint-filtering consistency. Indeed, kWC identifies inconsistent tuples inside the constraints that were initially accepted by them. Enforcing kWC amounts to reducing the constraints. It is based on the idea of extending tuples.

Definition 2. (Extension) *Let Y and Z be two sets of variables. A tuple τ' on $Y \cup Z$ is an* extension *on $Y \cup Z$ of a tuple τ on Y iff $\tau'[y] = \tau[y], \forall y \in Y$.*

Of course, a valid extension is simply an extension that corresponds to a valid tuple. We can now define k-wise consistency, which basically guarantees that every tuple in a constraint can be extended to any set of $k-1$ additional constraints. This kind of property allows us to reason about connections between constraints through shared variables.

Definition 3. *(*kWC) *A CSP $P = (X, D, C)$ is* k-wise consistent *(kWC) iff $\forall c_1 \in C, \forall \tau \in c_1 : \tau \in D(scope(c_1)), \forall c_2, \ldots, c_k \in C, \exists \tau'$ valid extension of τ on $\bigcup_{i=1}^{k} scope(c_k)$ satisfying $c_2, \ldots, c_k$.*

Note that k-wise consistency is called pairwise consistency for k=2 and three-wise consistency for k=3. It is immediate that k-wise consistency implies $(k-1)$-wise consistency. Enforcing kWC on a CSP involves removing from the constraints (i.e., considering as no longer allowed) the tuples that cannot be extended. It thus modifies the constraints, and not the domains as GAC does. As a result, kWC is incomparable with GAC: a CSP can be kWC but not GAC and reciprocally [JJNV89]. However, combining both consistencies allows us to make more pruning of the domains than GAC alone.

Definition 4. (GAC+kWC) *A CSP P is GAC+kWC iff P is both GAC and kWC.*

At this stage, although already suggested earlier, we can observe that GAC, kWC and GAC+kWC are well-behaved consistencies. We recall that a consistency ψ is *well-behaved* [Lec09] when for any CSP P, the ψ-closure of P exists, where the ψ-closure of P is the greatest CSP, denoted by $\psi(P)$, which is both ψ-consistent and equivalent to P. The underlying partial order on CSPs is: $P' = (X, D', C') \preceq P = (X, D, C)$ iff $\forall x \in X, D'(x) \subseteq D(x)$ and there exists a bijection μ from C to C' such that $\forall c \in C, \mu(c) \subseteq c$. Enforcing

ψ on P means computing $\psi(P)$, and an algorithm that enforces ψ is called a *ψ-algorithm*.

From GAC+kWC, we derive a domain-filtering consistency, called *domain k-wise consistency* (DkWC). When a CSP P is domain k-wise consistent, this means that none of the variable domains of P can be reduced when enforcing GAC+kWC.

Definition 5. (DkWC) *A CSP $P = (X, D, C)$ is* domain k-wise consistent *(DkWC) iff GAC+kWC(P) is a CSP $Q = (X, D^Q, C^Q)$ such that $D = D^Q$.*

GAC+kWC is both domain-filtering and constraint-filtering, which may render difficult its implementation in constraint solvers, whereas DkWC filters the domains and not the constraints. The pruning of DkWC is equivalent to that of GAC+kWC in the domains of the variables. The constraints are left unmodified by DkWC. Chapter 6 presents a filtering procedure for DkWC on table constrains which only relies on existing GAC propagators. GAC propagators exist in (almost) every CP solver. This makes the integration of DkWC into an existing solver very easy.

In the literature, GAC has already been combined with 2WC, 3WC and kWC [Jég91, JJNV89, BSW08, Ste08, KWR+10, Ste07, LXY14]. A first approach consists in weakening the combination, to obtain a pure domain-filtering consistency. We obtain then the max-restricted pairwise consistency (maxRPWC) [Ste07, Ste08, PS12].

Definition 6. (maxRPWC) *A CSP $P = (X, D, C)$ is max-restricted pairwise consistent (maxRPWC) iff $\forall x \in X$, $\forall a \in D(x)$, $\forall c \in C \mid x \in scope(c)$, $\exists \tau \in D(scope(c))$ such that $\tau[x] = a, \tau \in c$ and $\forall c' \in C$ there exists a valid extension of τ on $scope(c) \cup scope(c')$ satisfying c'.*

MaxRPWC is a domain-filtering consistency, close to the idea of D2WC and GAC+2WC but weaker than GAC+2WC [BSW08]. MaxRPWC is stronger than GAC. In [LPS13], the authors propose a specialized filtering procedure, called *eSTR*, for enforcing GAC+ 2WC (called full pairwise consistency in their paper) on table constraints. Two techniques are combined: simple tabular reduction (STR) and tuple counting. This allows eSTR to keep and update a counter, for each tuple of each table, of the number of supports it has in the other tables. This counter can be used to detect and remove unsupported tuples. This approach is orthogonal to the one that will be presented in Chapter 6. Indeed, in [LPS13], the authors lift up an existing GAC propagator, STR, to a propagator for GAC+2WC: eSTR. Our approach is to propose a filtering procedure, relying on a modified CSP, only using existing pure GAC propagators and we are not restricted to GAC+2WC. The idea of using GAC propagators to

enforce a consistency stronger than GAC has been used in [LXY14]. To obtain GAC+kWC, the authors use *factor variables*, representing variables that the constraints share, as well as additional constraints for $k \geq 3$. The fundamental difference between [LXY14] and the technique presented in Chapter 6 is that we only add one (dual) variable to the scope of each constraint, representing the constraint itself. In [LXY14], several variable can be added to the scope and represent the variables shared between constraints.

Other approaches, not weakening the combination of GAC and k-wise consistency, first compute the kWC-closure of a CSP and then apply GAC, as proposed in [Jég91, JJNV89, BSW08] for 2WC and [KWR+10] for kWC. The approach in [KWR+10] relies on a specialized propagator, first computing the k-wise consistent closure of the CSP and then obtaining GAC by projection onto the domains. The filtering process for the k-wise consistency inspects each constraint with respect to each relevant group of k constraints. Inspecting a constraint means searching, for each tuple of the constraint, for a support in each group. This search for a support is performed using a backtracking search (Forward Checking), on the dual encoding of the CSP. This whole process is sped up by memorizing, for each constraint and each group, the last encountered support. A similar approach is developed in [WKCB11] for relational neighborhood inverse consistency. In [KWR+10], the authors also propose a slightly weaker consistency, considering only groups of constraints forming connected components in the minimal dual graph. Although attractive, these original forms of propagators can be hard to include in existing constraint solvers. For instance, in *Comet*, the context management system makes the start of an independent search inside a propagator impossible.

Other related approaches exist, although not trying to directly enforce GAC+kWC. In [Lho12], the authors propose a consistent reformulation for the conjunction of two tables sharing more than one variable, keeping the spatial complexity low. In [BR99], the authors propose an algorithm to achieve GAC on global constraints. In that paper, the global constraints are perceived as groups of constraints and the CSPs they define are solved on the fly to achieve GAC on them. GAC+kWC on a group of k constraints can be seen as solving the subproblem they define, but in our approach, the subproblems are not solved on the fly.

The integration of strong levels of consistency into existing solvers has been studied in [VPJ11]. The integration is performed within a generic scheme, incorporating the subset of the constraints involved in the local consistency into a global constraint.

3

STATISTICAL TREATMENT OF THE EXPERIMENTAL DATA

When developing an algorithm, it is good practice to implement and test this algorithm on different benchmarks against other existing algorithms. This is the case for all the algorithms presented in this thesis. The results of such tests are measurements on the execution of the different algorithms put into competition, such as execution time, memory consumption, etc. Some algorithms can be better than an other on some benchmarks without being better on all of them. Moreover, some algorithms may have incomplete measurements. This is the case, for instance, when a limit is used on the resources (such as the execution time) and the algorithm fails to execute within these limits. This makes the comparison harder. In this context, we developed a procedure to compare such algorithms. This procedure has been developed in collaboration with Bernadette Govaerts and Cédric Heuchenne from the Institute of Statistics, Biostatistics and Actuarial Sciences (ISBA) of the *Université catholique de Louvain*. The statistical treatment of such experimental data is the topic of this chapter. Section 3.1 defines the context in which the procedure is to be used. Sections 3.2 and 3.3 then present the procedure itself. This procedure will be used, together with more traditional reporting, on all the experimentation made for this thesis.

3.1 PROBLEM DEFINITION

In this chapter, we will suppose that we have two algorithms, A and B. These algorithms are executed on *problem instances*. The measure of interest is here the execution time but the procedure can be applied to other measurements, such as the number of nodes, the number of failures of the search, etc. Each instance comes from a particular *problem class*. Classes can be, for instance, the problem itself (scheduling, car sequencing, etc.) but can also be a particularity of the instances (number of variables, number of constraints, etc.). Each class can contain a different number of instances and can have a different *importance*, which is user-defined. We will consider the different classes as *given* and the different instances in each class as a *sample* of the class full *population*. The data can contain *censored data*. Censored data occurs because the algorithm exceeds some fixed bound on the resource before completing. For execution time, censored data frequently occur in the presence of a timeout. Also, in this research, no hypothesis is made on the statistical distribution of the data. With such a decision to not make any hypothesis on the statistical distribution of the data, and in the presence of censored data, classical statistical tests such as the t-test or ANOVA cannot be used. Another particularity of the data we treat in this work is that it is *paired*. Indeed, algorithms A and B are executed on the same set of instances. The measurements for A on an instance and for B on the same instance are thus not independent.

The goal of this chapter is to propose a statistical treatment of the execution data to compare the performances of algorithms A and B. More precisely, we will be interested in generalizing to the whole population and all the classes, given their importances, a statistic of interest reflecting the relative performances of A and B. We thus define statistics of interest and rely on a well known method from applied statistics to provide confidence intervals on those statistics of interest.

Notation

We will consider two algorithms, A and B. We have k different classes, C_1, $C_2, \ldots, C_k$. For class j, the sample contains N_j instances. There is a total of N instances in the whole sample. Each instance y_i belongs to one class. The importance of class j is w_j. In the procedure, we will suppose that $\sum_{j=1}^{k} w_j = 1$. The measurements for instance y_i will be denoted by, respectively, $x_{A,i}$ and $x_{B,i}$ for algorithms A and B. The estimators for the statistics of interest are presented for a set of data. This set contains triplets $(y_i, x_{A,i}, x_{B,i})$ of the instance, the execution time for A, and the execution time for B.

3.2 TREATMENT OF DATA WITHOUT CENSORING

This section concerns the data without the presence of censoring. This means that the measurements made on the executions of the two algorithms are complete: the execution times for both algorithms are known for each instance. In this case, a special statistical treatment of the data is still presented because no hypothesis is made on the statistical distribution of the data. Also, the procedure for this case uses the same generalization method as the procedure for the case with censoring. The only difference between the two procedures lies in the statistic(s) of interest considered.

For this case, we define three statistics of interest: the arithmetic mean of differences (θ_1), the arithmetic mean of inverses of differences (θ_2), and the geometric mean of the ratios (θ_3). Their estimators ($\hat{\theta}$), presented below, are computed on the experimental results for a set of instances S, and for algorithms A and B. They are computed in such a way that a large positive number means A is better than B (smaller execution time) while a large negative number indicates that B is faster than A.

- **Arithmetic Mean of Difference**:

$$\begin{aligned}\hat{\theta}_1(A,B,S) = \quad & w_1 * \left(\sum_{\{y_i \in S : y_i \in C_1\}} \frac{x_{B,i} - x_{A,i}}{N_1}\right) + \dots \\ & + w_k * \left(\sum_{\{y_i \in S : y_i \in C_k\}} \frac{x_{B,i} - x_{A,i}}{N_k}\right)\end{aligned}$$

- **Arithmetic Mean of Inverses of Differences**:

$$\begin{aligned}\hat{\theta}_2(A,B,S) = \quad & w_1 * \left(\sum_{\{y_i \in S : y_i \in C_1\}} \frac{\frac{1}{x_{A,i}} - \frac{1}{x_{B,i}}}{N_1}\right) + \dots \\ & + w_k * \left(\sum_{\{y_i \in S : y_i \in C_k\}} \frac{\frac{1}{x_{A,i}} - \frac{1}{x_{B,i}}}{N_k}\right)\end{aligned}$$

- **Geometrical Mean of Ratios**:

$$\begin{aligned}\hat{\theta}_3(A,B,S) = \quad & \left(\prod_{\{y_i \in S : y_i \in C_1\}} \frac{x_{B,i}}{x_{A,i}}\right)^{\frac{w_1}{N_1}} * \dots \\ & * \left(\prod_{\{y_i \in S : y_i \in C_k\}} \frac{x_{B,i}}{x_{A,i}}\right)^{\frac{w_k}{N_k}}\end{aligned}$$

The difference between $\hat{\theta}_1$ and $\hat{\theta}_2$ is that $\hat{\theta}_2$ attaches more importance to short execution times. This can be useful for benchmarks with many small

solving times and some large ones. For $\hat{\theta}_3$, we are interested in ratios. As argued in [FW86], the geometric average is more accurate for ratios and normalized numbers. $\hat{\theta}_3$ thus uses the geometric mean. To summarize, the statistic of interest to use depends on the results and the desired quantity. If the execution times are comparable, or the long execution times are more important, and we are interested in the time difference, $\hat{\theta}_1$ is the appropriate measurement. If the benchmark results present short and long execution times, and we want the time difference to give importance to short execution times, $\hat{\theta}_2$ is the right choice. If we are interested in the ratios of performances of the algorithms, $\hat{\theta}_3$ is the appropriate statistic of interest.

The procedure used to estimate a confidence interval for the three defined statistics of interest is knows as the *basic bootstrap interval* [WPB00, Efr79, ET94, Dav97, You94]. The idea behind the basic bootstrap interval is the following.

Suppose a statistic of interest θ has to be computed on the cumulative distribution function F of a random variable X. F is unknown, as is the case for our experiments. An estimator $\hat{\theta}$ can be computed on the set of observed values of X: $O = \{x_1^*, x_2^*, \ldots, x_N^*\}$. $\hat{\theta}$ is thus a random variable itself. A confidence interval for the statistic of interest θ can be obtained using the probability distribution of $\hat{\theta} - \theta$: let s_α be the α-percentile of the distribution of $\hat{\theta} - \theta$, then we have that

$$P(s_{\alpha/2} \leq \hat{\theta} - \theta \leq s_{(1-\alpha/2)}) = 1 - \alpha$$

The confidence interval having probability $1 - \alpha$ to contain the true value of θ is thus:

$$\hat{\theta} - s_{(1-\alpha/2)} \leq \theta \leq \hat{\theta} - s_{\alpha/2}$$

As we make the assumption that F is unknown and no hypothesis is made on it, we do not know anything about the distribution of $\hat{\theta} - \theta$. This is where the bootstrap comes into play. By drawing M independent sets of observations from the set O with replacement and computing the estimator $\hat{\theta}$ on them, we can have observations of the random variable $\hat{\theta}$: $\hat{\theta}_1^*, \hat{\theta}_2^*, \ldots, \hat{\theta}_M^*$. With the assumption that the variation of $\hat{\theta}$ around θ is close to the variation of $\hat{\theta}^*$ around $\hat{\theta}$, we can derive a confidence interval for θ. This assumption is standard in the bootstrap community and often verified [Dav97]. If s_α^* represents the α-percentile of the distribution of $\hat{\theta}^* - \hat{\theta}$, the confidence interval with probability $1 - \alpha$ is estimated by:

$$\hat{\theta} - s_{(1-\alpha/2)}^* \leq \theta \leq \hat{\theta} - s_{\alpha/2}^*$$

It is easier to work directly with the percentiles of $\hat{\theta}^*$ than with those of $\hat{\theta}^* - \hat{\theta}$. If r^*_α is the α-percentile of the distribution of $\hat{\theta}^*$, we have

$$r^*_\alpha = s^*_\alpha + \hat{\theta}$$

Inserting this last relation into the confidence interval estimation, we have

$$2\hat{\theta} - r^*_{(1-\alpha/2)} \leq \theta \leq 2\hat{\theta} - r^*_{\alpha/2}$$

The α-percentile of $\hat{\theta}^*$ is given by the $(M \cdot \alpha)^{th}$ observed value in the ordered set of observations $\hat{\theta}^*_1 \leq \hat{\theta}^*_2 \leq \cdots \leq \hat{\theta}^*_M$.

For our algorithm execution measurements, the basic bootstrap procedure is the same for each statistic of interest defined. For a statistic of interest θ_s, this is summarized in Procedure 1. It consists in drawing, with replacement, M *bootstrap samples* (sets of instances) from among the entire sample regardless of the class, of size N (the same size as the original observation sample). The estimator of the statistic of interest is then computed on those M sets independently (taking classes into account through their importance). The basic bootstrap interval is then computed, with respect to the desired confidence level α.

In practice, these confidence intervals on the statistics of interest will be used to evaluate the differences between algorithms (pair by pair). If the confidence interval, with a significance level α, is $[i, j]$ for algorithms A and B, we know that the true statistic (θ, not $\hat{\theta}$) lies, with probability $1 - \alpha$, between i and j. We can thus quantify the difference between algorithms A and B with respect to the statistic of interest. This difference is said significant (with a significance level α) if 0 is outside the confidence interval for θ_1 and θ_2. For θ_3, it is significant if 1 is outside the confidence interval. We will thus say that algorithm A is significantly faster than B if the confidence interval for $\theta_1(A, B, S)$ or $\theta_2(A, B, S)$ (respectively, $\theta_3(A, B, S)$) is positive and does not contain 0 (respectively, 1). The bootstrap thus allows us to quantify the differences between algorithms and their significance.

3.3 TREATMENT OF DATA WITH CENSORING

The experimental data frequently contains censoring. Censoring occurs because the algorithms exceed a fixed bound on the use of the resources (total execution time, the number of nodes, the memory used, etc.). The different statistics from Section 3.2 cannot be used in this context. Indeed, for some i, either $x_{A,i}$ or $x_{B,i}$ is missing or both. Replacing them with the timeout value gives a non-precise test because the sets of instances for which data is censored

Basic Bootstrap Interval

1. Draw M independent bootstrap samples of size N.
   Each sample β_j is obtained by drawing N observations $(y_i^*, x_{A,i}^*, x_{B,i}^*)$ with replacement.
2. Compute the estimator of θ_s for each bootstrap sample: $\hat{\theta}_j^* = \hat{\theta}_s^*(A, B, \beta_j)$
3. Renumber the bootstrap replications of $\hat{\theta}$ such that:

$$\hat{\theta}_1^* \leq \hat{\theta}_2^* \leq \cdots \leq \hat{\theta}_M^*$$

4. The confidence interval is:

$$[2 * \hat{\theta}_{sample} - \hat{\theta}^*_{\lceil M \cdot (1-\alpha/2) \rceil}, 2 * \hat{\theta}_{sample} - \hat{\theta}^*_{\lfloor M \cdot \alpha/2 \rfloor}]$$

In the above procedure, $\hat{\theta}_{sample}$ is computed on the original sample and α is the desired significance level.

Procedure 1: Basic Bootstrap procedure for statistic of interest θ_s.

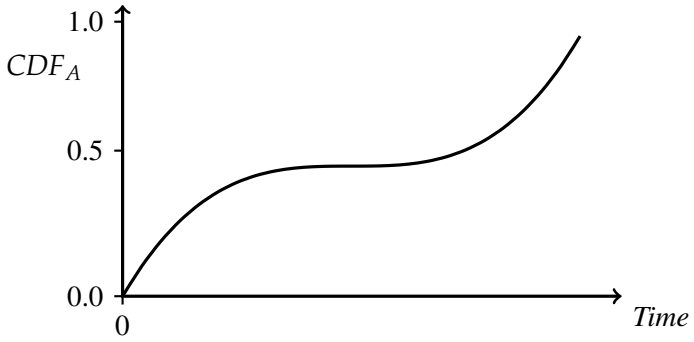


Figure 3.1: Example of a cumulative distribution function for an algorithm A

for A and B are most of the times distinct. As before, we will consider that the measurements are execution times and that censoring is present because of a timeout imposed on the experiments. The reasoning can be generalized to other kinds of measurements. We will present the procedure of the data for one instance class first, then we will generalize it to the multi-class setting. The only change made to the procedure in Section 3.2 is the statistic of interest used.

The tool we are using to treat data with censoring is the *cumulative distribution function* (CDF) of the solving times. This function gives, for an algorithm and each time point t, the probability for an instance to be solved in a time less than or equal to t. An example of a CDF is given in Figure 3.1.

Of course, we do not have access to the CDFs of the algorithms. We will thus use an estimator of the CDF: the *empirical cumulative distribution* (ECD). In the single class setting, the empirical cumulative distribution gives, at each time point t, the proportion of the instances for which the algorithm has an execution time less than or equal to t. Figure 3.2 gives an example of an empirical cumulative distribution. In this figure, the algorithm is able to solve 85% of the instances within the time limit TO. Formally, the definition of the empirical cumulative distribution is the following.

Definition 7. (Empirical Cumulative Distribution): *Let S be an instance set, A be an algorithm, and $x_{A,i}$ be the time required by A to solve instance $y_i \in S$ or $+\infty$ if A didn't solve instance y_i before the time limit. We have that the empirical cumulative distribution for A on S is defined as:*

$$ECD_A(t,S) = \frac{\#\{y_i \in S | x_{A,i} \leq t\}}{\#S},$$

where # is the cardinality of the set.

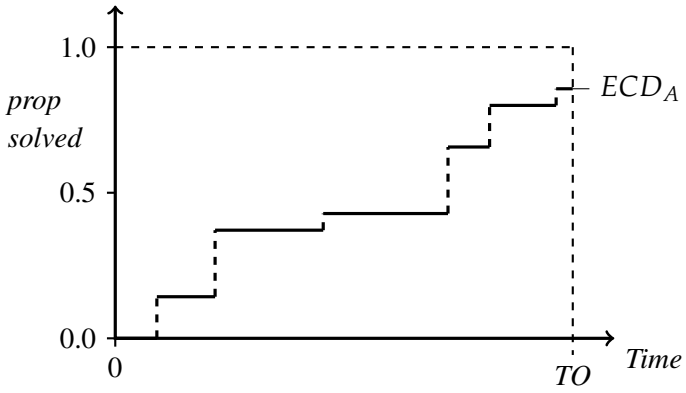


Figure 3.2: Example of an empirical cumulative distribution for an algorithm

Empirical cumulative distributions can be related to performance profiles [DM02]. For an algorithm A, its performance profile plots, for any ratio $r \geq 1$, the proportion of instances i for which the ratio of the time taken by A to solve i to the time taken by the best algorithm to solve i is less than or equal to r. The difference from the empirical cumulative distributions is the focus: performance profiles focus on ratios while empirical cumulative distributions focus on solving time. The choice of empirical cumulative distributions is motivated by our focus on solving time.

When comparing two algorithms, their empirical cumulative distributions can be used in different ways. For a time t between 0 and TO, the empirical cumulative distributions can give the difference in proportion of the instance set solved by each algorithm within this time budget t. We can also use this function the other way around. For a percentage q of the instance set, the empirical cumulative distributions of the algorithms can give the time needed by each algorithms to solve $q\%$ of the instances. Those two uses are illustrated in Figure 3.3. In Figure 3.3 (a), we have the proportion p_A of the instance set solved by A requiring less time than t and similarly for B and p_B. In Figure 3.3 (b) we have the times t_A and t_B taken respectively by A and B to solve $q\%$ of the instances.

A very important point about these graphs is that the set of instances considered for A and the one for B can be different. There is indeed no guarantee that the instances solved by A and B within a particular time budget are the same ones. It may seem like the pairing of the tests is lost in the empirical cumulative distributions but, as we will see later on, this is not (entirely) the case.

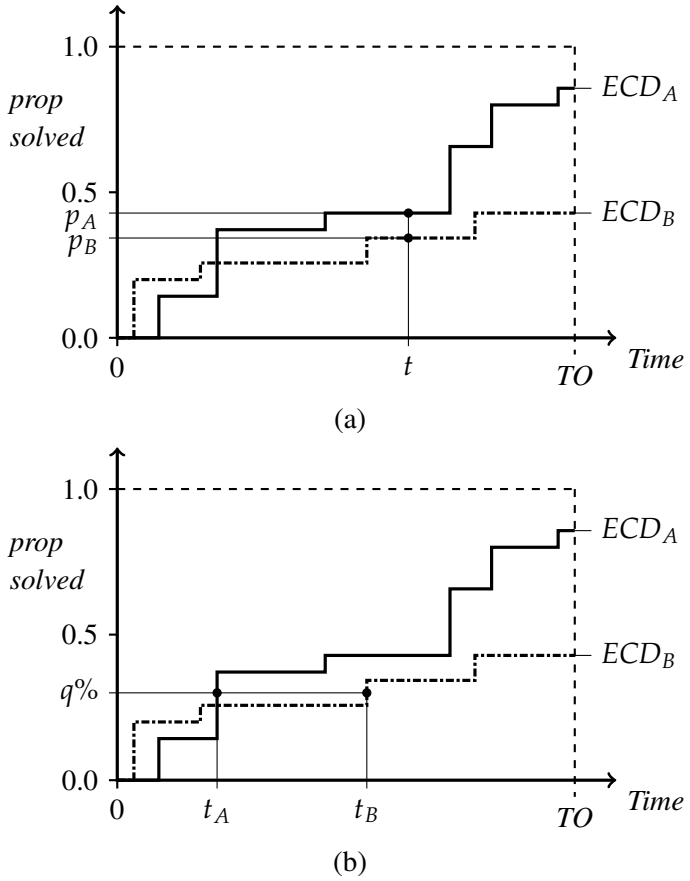


Figure 3.3: Utilisation of the empirical cumulative distributions for (a) the proportions p_A and p_B of the instance set solved after given time t and (b), the times t_A and t_B required to solve a given percentage of the instance set ($p\%$).

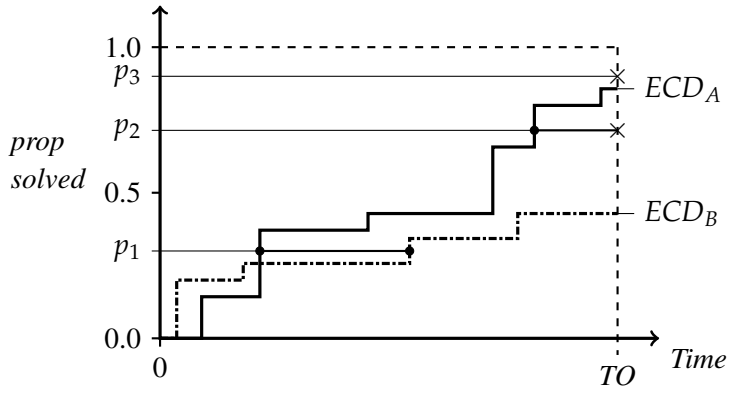


Figure 3.4: Example of the cumputability of the exact time differences for different proportions of the instance set: p_1 is computable, p_2 is partially computable and p_3 is not computable

As the measure of interest in this thesis is the execution time, we will be more interested in the setting of Figure 3.3 (b), where the time taken by the algorithms to solve a given proportion of the instance set is the measurement of interest. We can already notice that the exact time difference is not computable for some percentages. Figure 3.4 illustrates this problem. For p_1, the time difference is computable. For p_2, we can only have a lower bound on the real time difference and for p_3, we don't have any information.

Formally, we will thus make use of the inverse of the empirical cumulative distribution. Its definition under the presence of a time limit *TO* is given below.

Definition 8. (Inverse Empirical Cumulative Distribution)*: Let S be an instance set, A be an algorithm and ECD_A be its its empirical cumulative distribution. We define the Inverse Empirical Cumulative Distribution as:*

$$ECD_A^{-1}(p,S) = \begin{cases} \textbf{if}(\nexists t : 0 \leq t \leq TO \wedge ECD_A(t,S) \geq p) : \perp \\ \\ \textbf{else} : min_{\{t:0\leq t\leq TO\wedge ECD_A(t,S)\geq p\}}(t) \end{cases}$$

We are not interested in solving a particular proportion of the instance set but we would like to assess the performances of the algorithms in general. The algorithms being compared are meant to be used on an unknown number of unknown instances and the time budget that will be given to them is also unknown. The measurement we use in this setting is thus the mean time difference between the algorithms for the different proportions of the instance set

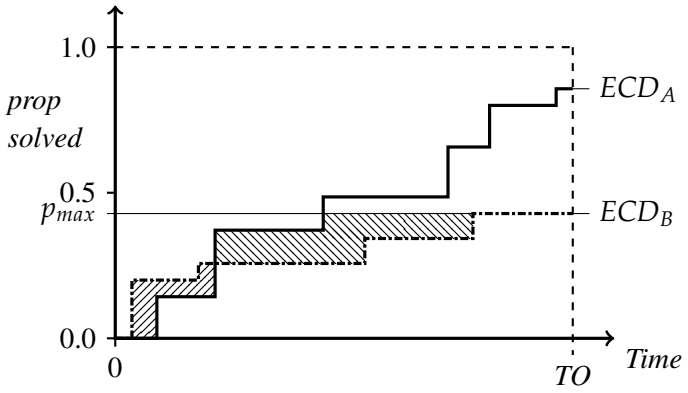


Figure 3.5: Illustration of the area where the exact time differences are computable.

where the exact time differences are computable. This can be seen as the mean width of the area between the empirical cumulative distributions where the time differences are computable. This area is illustrated in Figure 3.5. In this plot, the areas before and after the crossing point of the ECDs have opposite signs.

As we only consider the area where the exact time differences are computable, the definition of our measurement makes use of the following quantity.

Definition 9. ($p_{max}(A,B,S)$)*: Let A and B be two algorithms with respective empirical cumulative distributions ECD_A and ECD_B and S be an instance set. We define the maximum computable proportion as:*

$$p_{max}(A,B,S) = \mathbf{min}(ECD_A(TO,S), ECD_B(TO,S))$$

This definition captures the maximum proportion such that all the time differences below it are exactly computable. Figure 3.5 illustrates p_{max}. For an instance set S (one class), in the presence of censored data, the estimator of the statistic of interest is, for algorithms A and B:

$$\hat{\theta}_4(A,B,S) = \frac{1}{p_{max}(A,B,S)} \int_0^{p_{max}(A,B,S)} (ECD_B^{-1}(p,S) - ECD_A^{-1}(p,S))dp$$

Notice that we subtract ECD_A^{-1} from ECD_B^{-1}. This captures the fact that for a given proportion of the instances, a smaller solving time is better. This thus makes $\hat{\theta}_4(A,B,S)$ positive if A is better than B. $\hat{\theta}_4$ is also normalized with respect to $p_{max}(A,B,S)$ to get the mean time difference between the

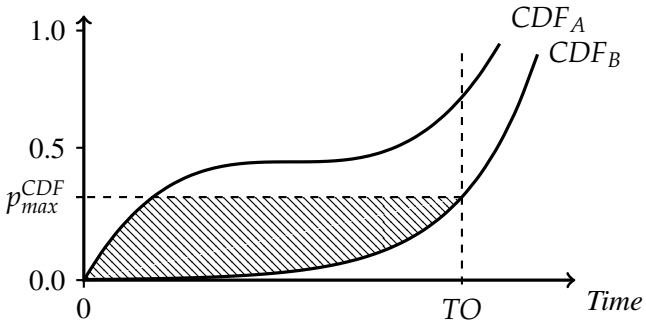


Figure 3.6: Illustration of the statistic of interest θ_4

algorithms when considering all the proportions of the instance set from 0 to p_{max}. This makes $\hat{\theta}_4$ independent of p_{max}. If there are no timeouts in the instance set, the quantity $\hat{\theta}_4(A, B, S)$ is exactly $\hat{\theta}_1(A, B, S)$.

The basic bootstrap procedure defined in Section 3.2, Procedure 1, is then used on $\hat{\theta}_4$. This is where some of the pairing of the data is recovered. Indeed, instances (and not solving times) are drawn at random from the set of instances. This means that if an instance is drawn, it will be taken into account for the computations of empirical cumulative distributions of both algorithms.

This measurement $\hat{\theta}_4$ is an estimator of the mean width of the area between the real cumulative distribution functions of the algorithms in the presence of a time limit, as shown in Figure 3.6.

In the multi-class setting, inside an instance set S we have different classes of instances $C_1, \ldots, C_k$. Each class C_j has a predefined importance w_j (such that $(\sum_{j\in\{1\ldots k\}} w_j) = 1$). The measurement we define in this case ($\hat{\theta}_5$) is very similar to $\hat{\theta}_4$. Only the empirical cumulative distributions are changed to reflect the importances of the instance classes. We thus define the multi-class empirical cumulative distribution:

Definition 10. (Multiclass Empirical Cumulative Distribution): *Let S be an instance set, A be an algorithm, and $x_{A,i}$ be the time required by A to solve instance $y_i \in S$ or $+\infty$ if A didn't solve instance y_i before the time limit. We have that the empirical cumulative distribution for A on S is defined as:*

$$MECD_A(t, S) = \sum_{j\in\{1\ldots k\}} \left[w_j * \frac{\#\{y_i \in S : y_i \in C_j \wedge x_{A,i} \leq t\}}{\#\{y_i \in S : y_i \in C_j\}} \right]$$

where # is the cardinality of the set.

In ECD_A, at each time point, the value of the function is the proportion of the entire set that has been solved. Here the proportions in the different classes

are simply aggregated using their different importances. $MECD^{-1}$ is defined similarly to ECD^{-1} and mp_{max} is defined similarly to p_{max}. The measurement for the multi-class setting in the presence of censoring is then:

$$\hat{\theta}_5(A,B,S) = \frac{1}{mp_{max}(A,B,S)} \int_0^{mp_{max}(A,B,S)} (\quad MECD_B^{-1}(p,S) \\ -MECD_A^{-1}(p,S))dp$$

The bootstrap procedure defined in Section 3.2 is then also be used on $\hat{\theta}_5$. Similarly to $\hat{\theta}_4$, $\hat{\theta}_5$ is an estimator of the mean width of the area between the multi-class versions of the CDFs of the algorithms under the presence of a time limit.

Part II

PROPAGATION FOR TABLE CONSTRAINTS

4

EFFICIENT AND OPTIMAL GAC PROPAGATORS FOR TABLE CONSTRAINTS

As explained in Chapter 1, the table constraint is a core constraint. By listing the allowed combinations of values for its variables, it permits encoding any constraint. Chapter 2 explained the importance of propagation in constraint programming, allowing substantial reductions of an exponentially large search tree. This chapter presents five different generalized arc consistency (GAC) propagators for table constraints. In other words, this chapter presents the adaptation of an important mechanism for a core constraint. Due to its form, the propagation of table constraints cannot make use of particular knowledge of the constraint and must deal with the raw list of allowed tuples. This chapter thus presents effective and optimal means to use this raw list of allowed tuples to reduce the domains of the variables. By optimal, we mean that the worst case time complexity of the algorithm is minimal.

It starts by presenting AC5, the framework used by our propagators. Then it presents efficient but non-optimal propagators. Afterward, it presents a variation of our algorithm based on recomputation of information rather than storage before presenting the optimal propagators and the experimental results of all the presented algorithms. Although presented in the context of CSPs, these

algorithms can of course be used in constraint optimization problems, as the propagation of the constraints is orthogonal to the optimization of the objective.

Related Publications

[MVHD12] Jean-Baptiste Mairy, Pascal Van Hentenryck and Yves Deville, "An Optimal Filtering Algorithm for Table Constraints", 18th International Conference on Principles and Practice of Constraint Programming (CP 2012), 2012, Québec City, Canada.

[MVHD14a] Jean-Baptiste Mairy, Pascal Van Hentenryck and Yves Deville, "Optimal and Efficient Filtering Algorithms for Table Constraints", Constraints 19 (1), pages 77-120

4.1 THE AC5 ALGORITHM

The algorithms presented in this chapter all use the value-based paradigm. In a constraint-based approach, the propagation queue contains information about the constraints that need to re-enforce consistency. In a value-based approach, information on the removed values is also stored in the queue for the propagation. The propagation algorithms are called once for each deleted literal. Their job is then to reflect the deletion of that literal. GAC considers constraints in isolation. To obtain GAC on all constraints at the same time, a fixed-point algorithm is needed. Indeed, if a constraint removes a literal, this might compromise the consistency of the other constraints. The generic fixed-point algorithm using the value-based propagation paradigm chosen to embed the propagators designed in this chapter is AC5 [VDT92, DV10]. AC5 uses a queue Q of triplets (c, x, a) stating that the domain consistency of constraint c should be reconsidered because value a has been removed from $D(x)$. For the presentation and specification of AC5 (and other algorithms in this chapter), the following sets are useful. Let c be a constraint with arity r, of a CSP (X, D, C) with $y \in scope(c)$, and B be some domain for the variables X.

$$Inc(c, B) = \{(x, a) |\ x \in scope(c) \wedge a \in D(x) \wedge \forall \tau \in B(scope(c))_{x=a} : \neg c(\tau)\}$$

$$Cons(c, y, b) = \{(x, a) | x \in scope(c) \wedge\ a \in D(x) \wedge \exists \tau \in c : \tau[x] = a\ \wedge \tau[y] = b\}$$

$$Inc(c) = Inc(c, D)$$

$Inc(c, B)$ represents the set of domain inconsistent literals of constraint c with respect to domain B. $Cons(c, y, b)$ is the set of literals in the tuples allowed by c having value b for variable y. A constraint c in a CSP (X, D, C) is *domain-consistent* iff $Inc(c) = \emptyset$. A CSP (X, D, C) is *domain-consistent* iff all its constraints are domain-consistent.

Specification 1 describes the main methods of AC5. In the postcondition of `enqueue`, Q_o represents the value of Q at call time. The propagators using AC5 should define their own `post` and `valRemove` methods. The generic AC5 algorithm, using those methods, is depicted in Algorithm 1. In all the pseudocodes presented in this chapter, the assumed context is the resolution of a CSP (X, D, C) and a propagation queue Q. The working principle of AC5 consists of two parts: initialization (`initAC5`) and queue propagation (`propagateQueueAC5`). In the initialization, the `post`$(c, \triangle)$ method is called once for each constraint c. Its role is to compute the inconsistent values of the constraint and to initialize specific data structures required for the propagation. Each time a value is removed from a domain, `enqueue` puts the necessary information in the propagation queue. In the second phase of AC5, while there are triplets (c, y, b) in the queue, `valRemove`(c, y, b) is called so that the constraint c can reflect the removal of b from $D(y)$, possibly removing more literals. This dequeuing/enqueuing process is repeated until the queue becomes empty. At this point, the constraints using AC5 are generalized arc consistent. As long as (c, x, a) is in the queue, it is algorithmically desirable to consider that value a is still in $D(x)$ from the perspective of constraint c. This is captured by the following definition.

Definition 11. *The local view of a domain $D(x)$ wrt a queue Q for a constraint c is defined as $D(x, Q, c) = D(x) \cup \{a | (c, x, a) \in Q\}$.*

For a constraint c, a queue Q and a set of variables $X = \{x_1 \dots x_n\}$, $D(X, Q, c)$ is defined as $\{D(x_1, Q, c), \dots, D(x_n, Q, c)\}$. In a table constraint c, a tuple τ is *Q-valid* if all its values belong to $D(scope(c), Q, c)$. The central method of AC5 is the `valRemove` method, where the set $\triangle$ is the set of values becoming inconsistent because b is removed from $D(y)$. In this specification, b is a value that is no longer in $D(y)$ and `valRemove` computes the values (x, a) no longer supported in the constraint c because of the removal of b from $D(y)$. Note that values in the queue are still considered in the potential supports as their removal has not yet been reflected in this constraint. The minimal pruning $\triangle_1$ only deals with variables and values previously supported by (y, b). However, we give `valRemove` the ability to achieve more pruning ($\triangle_2$), which is useful for table constraints.

```
1  enqueue(in x: Variable; in a: Value;
2          in C1: Set of Constraints; inout Q: Queue)
3  // Pre: x ∈ X, a ∉ D(x), C1 ⊆ C
4  // Post: Q = Q0 ∪ {(c,x,a)|c ∈ C1, x ∈ scope(c)}
5
6  post(in c: Constraint; out △: Set of Values)
7  // Pre: c ∈ C
8  // Post: △ = Inc(c) + initialization of specific data structures
9
10 valRemove(in c: Constraint; in y: Variable;
11           in b: Value; out △: Set of Values)
12 // Pre: c ∈ C, b ∉ D(y,Q,c)
13 // Post: △1 ⊆ △ ⊆ △2 with △1 = Inc(c,D(X,Q,c)) ∩ Cons(c,y,b)
14 //                    and △2 = Inc(c)
```

Specification 1: The enqueue, post, and valRemove Methods for AC5

```
1  AC5(in X, C, inout D){
2  // Pre: (X,D,C) is a CSP
3  // Post: D ⊆ D0, (X,D,C) equivalent to (X,D0,C)
4  //       (X,D,C) is domain consistent
5    initAC5(Q);
6    propagateQueueAC5(Q);
7  }

8  initAC5(out Q){
9   Q = ∅;
10  C1 = ∅;
11  forall(c in C){
12   post(c,△);
13   forall((x,a) in △){
14    D(x) -= a;
15    enqueue(x,a,C1,Q);
16   }
17  C1 += c;
18  }
19 }

20 propagateQueueAC5(in Q){
21  while Q != ∅ {
22   select (c,y,b) in Q;
23   Q = Q - (c,y,b);
24   valRemove(c,y,b,△);
25   forall((x,a) in △){
26    D(x) -= a;
27    enqueue(x,a,C\{c},Q);
28   }
29  }
30 }
```

Algorithm 1: The AC5 Algorithm.

The AC5 algorithm may seem to be dependent on a particular solver. Indeed, propagation queues and fixed-point algorithms implementing the AC5 framework are not present in every solver. For instance, AC5 is implemented in Comet and OscaR[1] but not in GeCode[2]. However, many solvers provide, when a propagator is called, the information on the changes that occurred in the domains of the variables since the last call to the propagator (called deltas). In those solvers, not implementing AC5 but having those deltas, it is easy to implement our propagators. Indeed, to simulate AC5, it suffices to treat the literals in the deltas one by one with the same implementation of `valRemove`. For the solvers neither implementing AC5 nor the deltas, our algorithms can still be implemented. In such a case, it would be the constraint's responsibility to compute the deltas. This is possible by recording, at the end of each call to the propagator, the current state of the domains. In this way, the deltas can be computed at the beginning of the next call to the propagator. Unfortunately, this would have an additional cost since the current state of the domains has to be maintained during the search.

Notations Throughout this chapter, we will use the implicit ordering of the tuples inside table constraints: $\tau_{c,i}$ denotes the i^{th} element of the table in the table constraint c and $\tau_{c,i}[x]$ is the value of $\tau_{c,i}$ for variable x. We introduce a top index $\top$ (respectively, bottom index $\bot$) greater (respectively, smaller) than any other index. We also introduce a universal tuple $\tau_{c,\top}$, with $\tau_{c,\top}[x] = *$ forall $x \in X$ and abuse notation in postulating that $\forall a \in D(x), * = a$. This universal tuple can thus be found in any table. More precisely, for any table T, $\tau_{c,\top} \in T$.

4.2 EFFICIENT GAC PROPAGATORS

GAC is based on the notion of support. An algorithm computing GAC has to remove the literals that do not have a support. One possible technique to do so is to memorize the first support of each literal and to update it when this support becomes invalid. When no more valid support can be found, the literal can be removed. The algorithms defined in this chapter use a data structure FS memorizing first supports. Intuitively, $FS[x,a]$ is the index of the first Q-valid support of (x,a). It is thus equivalent to the *last* structure used in GAC2001 algorithms. To speed up the table traversal, our algorithms use a second data structure called *next* that links together all the tuples of the table sharing the

1 http://oscarlib.bitbucket.org/
2 http://www.gecode.org/

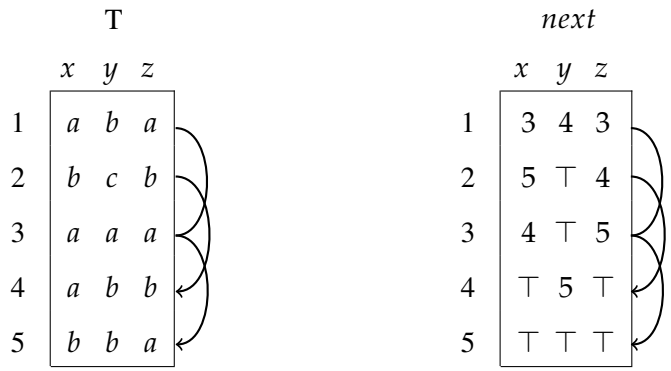


Figure 4.1: Example of a *next* data structure of a table T (arrow pointers for variable z only).

same value for a given variable. For a tuple index and a variable, *next* gives the index of the next tuple having the same value for the variable. More formally, for a given table constraint c, *FS* and *next* satisfy the following invariant (called the FS-invariant) before dequeuing an element from Q.

$$\begin{aligned}
&\forall x \in scope(c)\ \forall a \in D(x,Q,c) : FS[x,a] = i \Leftrightarrow \\
&\quad \tau_{c,i}[x] = a \wedge i \neq \top \wedge \tau_{c,i} \in D(scope(c),Q,c) \\
&\quad \wedge \forall j < i : \tau_{c,j}[x] = a \Rightarrow \tau_{c,j} \notin D(scope(c),Q,c) \\
&\forall x \in scope(c)\ \forall\, 1 \leq i \leq c.length : \\
&\quad next[x,i] = Min\{j | i < j \wedge \tau_{c,j}[x] = \tau_{c,i}[x]\}
\end{aligned}$$

The *next* data structure, illustrated in Figure 4.1, is static as it does not depend on the domain of the variables. However, *FS* must be trailed during the search.

Propagators using the AC5 framework only need to define their own `post` and `valRemove` methods. The two first efficient propagators of this chapter use the same structure for their `post` and `valRemove` methods. Methods `postTC` and `valRemoveTC` are thus given in Algorithms 2 and 3. TC is included in the method and algorithm names to underline the fact that those are propagators for Table Constraints. Note that the $\triangle$ computed by `valRemoveTC` corresponds to $\triangle_1$ in Specification 1. Method `valRemoveTC` uses the `seekNextSupportTC` method (Algorithm 4), which searches for the next Q-valid support for a literal. The abstract method `isQValidTC(c,i)` tests whether $\tau_{c,i}$ is Q-valid (i.e., $\tau_{c,i} \in D(scope(c),Q,c)$) and can be implemented in many ways. One simple way is to record the Q-validity of tuples in some

data structure, initialized in method `initSpecStructTC` and updated in method `setQInvalidTC`. Method `postTC` initializes the FS and *next* data structures and returns the set of inconsistent values. Method `valRemoveTC` only has to consider the tuples in the *next* chain for (y,b) from $FS[y,b]$. Those are all the newly Q-invalid tuples. The tuples before are invalid and cannot be the first support of any other Q-valid literal. The tuples outside the *next* chain for (y,b) do not contain (y,b). When one of the traversed tuples $\tau_{c,i}$ is the first support of an element $a = \tau_{c,i}[x]$, a new support $FS[x,a]$ must be found. Indeed, $\tau_{c,i}$ is no longer Q-valid. If such a new support does not exist, then (x,a) belongs to the set $\triangle$. Method `valRemoveTC` thus computes the set $\triangle$ and maintains the FS-invariant. The AC5 algorithm with the `postTC` and `valRemoveTC` implementation for table constraint is called AC5TC (AC5 for Table Constraints).

```
1  postTC(in c: Constraint;out △: Set of Values){
2  // Pre: c ∈ C, c is a table constraint
3  // Post: △ = Inc(c) + initialization of the next, FS and
4  //              specific data structures
5      △ = ∅;
6      initSpecStructTC(c);
7      forall(x in scope(c), a in D(x)) c.FS[x,a]=⊤;
8      forall(x in scope(c), i in 1..c.length)  c.next[x,i]=⊤;
9      forall(i in c.length..1)
10         if (τ_{c,i} in D(scope(c))){
11             forall(x in scope(c)){
12               c.next[x,i] = FS[x,τ_{c,i}[x]];
13               c.FS[x,τ_{c,i}[x]] = i;
14             }
15         }
16         else setQInvalidTC(c,i);
17     forall(x in scope(c),a in D(x))
18         if(c.FS[x,a]==⊤) △ += (x,a);
19 }
```

Algorithm 2: Method `postTC` for Table Constraints

We will now analyze the complexity of AC5TC. None of the analyses of the complexities of the propagation presented in this thesis will include the complexity of handling the backtrackable structures. The complexity analysis thus does not include the time complexity of storing information about the modification of the data structures, nor the complexity of restoring them on backtracking. The complexity analysis thus represents the complexity of performing the pruning at one node. This complexity, as we put aside the cost

```
1  valRemoveTC(in c: Constraint;in y: Variable;
2           in b: Value; out △: Set of Values) {
3  // Pre: c ∈ C, c is a table constraint and b ∉ D(y,Q,c)
4  // Post: △ = Inc(c,D(X,Q,c)) ∩ Cons(c,y,b)
5      △ = ∅;
6      i = c.FS[y,b];
7      while(i!=⊤){
8          setQInvalidTC(c,i);
9          forall(x in scope(c): x!=y){
10             a = τ_{c,i}[x];
11             if (c.FS[x,a]==i){
12                 c.FS[x,a] = seekNextSupportTC(c,x,i);
13                 if(c.FS[x,a]==⊤ && a in D(x))  △ += (x,a);
14             }
15         }
16         i = c.next[y,i];
17     }
18 }
```

Algorithm 3: Method `valRemoveTC` for Table Constraints.

```
1  seekNextSupportTC(in c: Constraint; in x: Variable;
2                    in i: Index) : Index {
3  // Pre: c ∈ C, c is a table constraint, x ∈ scope(c), 1 ≤ i ≤ c.length
4  // Post: return the first index j greater than i of a Q−valid tuple with
5  //       τ_{c,j}[x] == τ_{c,i}[x]
6      i = c.next[x,i];
7      while(i!=⊤){
8          if(isQValidTC(c,i))  return i;
9          i = c.next[x,i];
10     }
11     return ⊤;
12 }
```

Algorithm 4: Function `seekNextSupportTC` for Table Constraints.

of storing backtracking information, corresponds to the complexity of enforcing the consistency along a descending path in the search tree. The reason to not include those costs is the following: the way backtrackable information is handled is solver-dependent and the costs differ from one solver to one another. For instance, in comet, trailing is implemented. With trailing, the cost of backtracking a structure is proportional to the number of modifications in the structure. In Gecode, backtracking is implemented with a copy, at each node, of the backtrackable quantities. The cost is of course not the same.

Proposition 1. *Assuming that* `initSpecStructTC` *and* `setQInvalidTC` *have a time complexity of* $O(r \cdot t + r \cdot d)$ *and* $O(1)$ *respectively and allow a correct implementation of* `isQValidTC` *to have a complexity of* $O(r)$*, then AC5TC is correct and has a time complexity of* $O(r^2 \cdot t + r \cdot d)$ *per table constraint along a branch in the search tree.*

Proof. Assuming a correct implementation of `isQValidTC`, AC5TC is correct. Indeed, `postTC` and `valRemoveTC` respect their specification (Specification 1). Not considering `initSpecStructTC`, method `postTC` has a time complexity of $O(r \cdot t + r \cdot d)$. After the `postTC` method, the domain size of x is $O(t)$ since each value in $D(x)$ has at least one support in the table. We now establish the complexity of all executions of `valRemoveTC` for a given table constraint, assuming this table constraint is one of the constraints of the CSP on which domain consistency is achieved. Consider first all executions of `valRemoveTC` without line 12. For a given variable y, these executions follow the different *next* chains of the variable y. The chains for all values of y have a total number of t elements. The complexity of lines 9–16 (without line 12) is $O(r)$. Since the table has r variables, the complexity of all `valRemoveTC` executions during the fixed point (without line 12) is thus $O(r^2 \cdot t)$, assuming an $O(1)$ complexity of `setQInvalidTC`. Consider now all executions of line 12 in `valRemoveTC` for a variable x. Since line 12 always increases the value of $FS[x,a]$ in the *next* chain of (x,a), we have a global complexity of $O(V \cdot t)$ for the variable x, where V is the time complexity of `isQValidTC`. All executions of line 12 in `valRemoveTC` thus take time $O(V \cdot r \cdot t)$. The time complexity of all executions of `valRemoveTC` is then $O(r^2 \cdot t + V \cdot r \cdot t)$. With an $O(r)$ `isQvalidTC`, this complexity is $O(r^2 \cdot t)$, giving AC5TC a complexity of $O(r^2 \cdot t + r \cdot d)$ per table constraint along a branch in the search tree. □

To achieve domain consistency, one must at least check the validity of each tuple and, in the worst case, remove all the values from the domains. Hence a domain-consistency algorithm has a complexity $\Omega(r \cdot t + r \cdot d)$ per table

constraint in the worst case. An AC5-like algorithm with a complexity $O(r \cdot t + r \cdot d)$ per table constraint is thus optimal. For most incremental propagators, if the time complexity for obtaining domain consistency is $O(f)$, then the time complexity of the all executions of this algorithm along any descending path in the search tree is also $O(f)$, ignoring the cost imputable to backtrackable structures. Indeed, the worst case scenario at one node is the removal of all literals one by one. As literals can only be removed once in any descending path of the search tree, this is also the worst case scenario for a descending path in the search tree. Even with an $O(1)$ the time complexity of `isQValidTC`, AC5TC has a complexity of $O(r^2 \cdot t + r \cdot d)$ per table constraint along a branch in the search tree. Thus, it does not have the optimal time complexity, but its implementations turn out, in the experiments, to be more efficient than state-of-the-art algorithms on some classes of problems.

```
1  initSpecStructTC-Bool(in c: Constraint) {
2      forall(i in 1..c.length)  c.isQValid[i] = true;
3  }
4
5  isQValidTC-Bool(in c:Constraint;in i:Index){
6  // Pre: c ∈ C, c is a table constraint and 1 ≤ i ≤ c.length
7  // Post: returns τ_{c,i} ∈ D(X,Q,c)
8      return c.isQValid[i];
9  }
10
11 setQInvalidTC-Bool(in c: Constraint;in i: Index) {
12 // Pre: c ∈ C, c is a table constraint and 1 ≤ i ≤ c.length
13     c.isQValid[i] = false;
14 }
```

Algorithm 5: Implementation of the specific methods of AC5TC-Bool

We now present two implementations of AC5TC. They differ in how they implement `isQValidTC`, `setQInvalidTC` and `initSpecStructTC`. The specific methods of AC5TC-Bool, the first implementation of AC5TC, are shown in Algorithm 5. AC5TC-Bool uses a data structure $isQValid[i]$ to record the Q-validity of the element $\tau_{c,i}$. It satisfies the invariant $isQValid[i] \Leftrightarrow \tau_{c,i} \in D(scope(c), Q, c)$ before dequeuing an element from Q ($1 \leq i \leq c.length$). The data structure must be backtracked during search, as it depends on the domains. Indeed, as the domains are restored on backtracking during the search, this structure must be, too. As the specific methods of AC5TC-Bool are correct, AC5TC-Bool is correct. The time complexity of `isQValidTC-Bool` and `setQInvalidTC-Bool` is $O(1)$ and `initSpecStructTC-Bool` is

```
1  initSpecStructTC-Sparse(in c: Constraint){
2     forall(i in 1..c.length){
3        c.Map[i] = i;
4        c.Dyn[i] = i;
5     }
6     c.size = c.length;
7  }
8
9  isQValidTC-Sparse(in c: Constraint;
10                              in i: Index;out b: Bool){
11 // Pre: c ∈ C, c is a table constraint and 1 ≤ i ≤ c.length
12 // Post: return (τ_{c,i} ∈ D(X,Q,c))
13       return (c.Map[i] <= c.size);
14 }
15
16 setQInvalidTC-Sparse(in c: Constraint; in i: Index){
17 // Pre: c ∈ C, c is a table constraint and 1 ≤ i ≤ c.length
18    c.Dyn[c.Map[i]] = c.Dyn[c.size];
19    c.Dyn[c.size] = i;
20    c.Map[c.Dyn[c.Map[i]]] = c.Map[i];
21    c.Map[i] = c.size;
22    c.size--;
23 }
```

Algorithm 6: Implementation of the specific methods of AC5TC-Sparse

$O(t)$. The time complexity of AC5TC-Bool is then $O(r^2 \cdot t + r \cdot d)$ per table constraint.

AC5TC-Bool must backtrack the *isQValid* Boolean array. Backtracking this array during the search is costly. During the search, when moving from one search node to another one, this array must either be restored to its previous state or information on the modifications of the array must be stored. We thus now propose an implementation that only trails one integer, building upon an idea in STR, STR2 and STR2+ [Ull07, Lec11] originally described in [BT93]. Applied to a table, this structure, called Sparse Set, keeps the elements that have been removed at the end of the table, with a single variable *size* representing the boundary between present (before position *size*) and removed elements (after position *size*). When an element is removed, it is swapped with the element at position *size*, and *size* is decremented by one. This representation uses two arrays *Map* and *Dyn* to represent the sparse set. For each entry of the set, *Map* contains its position in the array *Dyn*—this array contains the elements of the set before the index *size*, and the ones which have been removed are after *size*. Figure 4.2 depicts the structures representing the set $\{1,2,3,4,6,7,9,10\}$ from which 5 and 8 had previously been removed. Here, as will be the case in our propagator, the values are the successive integer values between 1 and n. This is not a necessary condition. Figure 4.3 illustrates the remove operation. The *size* variable must be backtracked during search (again, because it depends on the domains) but the arrays *Map* and *Dyn* do not need to be. On backtrack, restoring *size* automatically restores the elements of the table, albeit at a different position in the table. This is sometimes called semantic backtracking [VHR95]. For this implementation, the Sparse Sets are used to keep track of the set of Q-valid tuple indexes in the table of the constraint. Figure 4.4 illustrates the use of the sparse sets for this implementation. Instead of trailing t Booleans as AC5TC-Bool is doing, AC5TC-Sparse only has to trail *size* (one integer). More formally, for a given table constraint c, the data structures satisfy the following invariants before dequeuing an element from Q:

$$\forall\, 1 \leq i \leq c.length :$$
$$(Map[i] \leq size \Leftrightarrow \tau_i \in D(scope(c), Q, c)\ ) \wedge\ Dyn[Map[i]] = i$$

Testing the Q-validity of a tuple index i in such set is easy: it suffices to test whether $Map[i]$ is less than or equal to *size*. This is an $O(1)$ operation.

The implementations of the specific functions for this algorithm are given in Algorithm 6: it is called AC5TC-Sparse. The time complexity of `isQValid-TC-Sparse` and `setQInvalidTC-Sparse` is $O(1)$. The time complexity

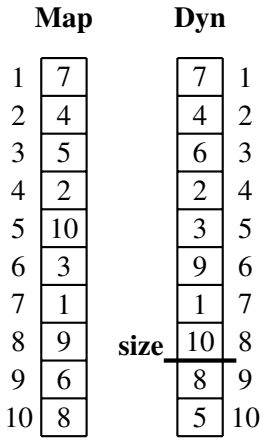
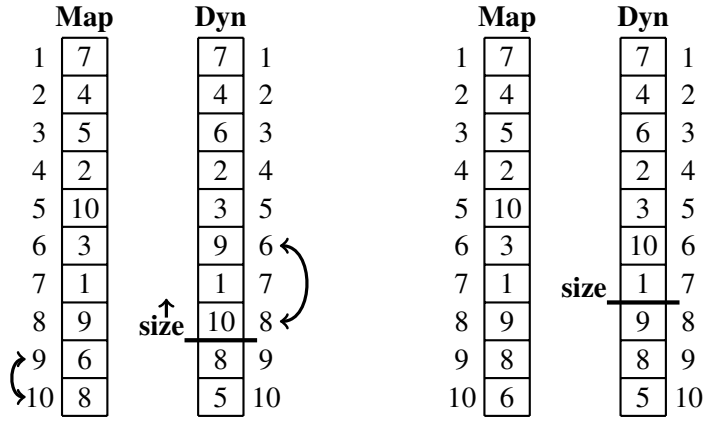


Figure 4.2: The two arrays representing the Sparse Set $\{1, 2, 3, 4, 6, 7, 9, 10\}$



(a) Operations to remove 9 from the set

(b) Resulting *Dyn* and *Map*

Figure 4.3: Removing 9 from the set $\{1, 2, 3, 4, 6, 7, 9, 10\}$ (5 and 8 previously removed)

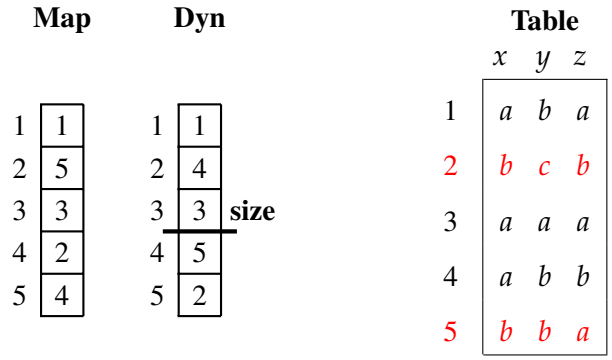


Figure 4.4: AC5TC-Sparse using Sparse sets to keep track of Q-valid tuples. In this example, tuples 2 and 5 are Q-invalid.

of `initSpecStructTC-Sparse` is $O(t)$. The time complexity of AC5TC-Sparse is thus $O(r^2 \cdot t + r \cdot d)$.

The spatial complexities of both AC5TC-Bool and AC5TC-Sparse are $\Theta(r \cdot t + r \cdot d)$. Indeed, the *next* structure is $\Theta(r \cdot t)$ since each tuple appears in r chains, *FS* is $\Theta(r \cdot d)$ and both *isQValid* and *Map*/*Dyn* are $\Theta(t)$. AC5TC-Bool has $\Theta(r \cdot d)$ integers and $\Theta(t)$ Booleans in its data structure to maintain during the search: it must maintain *FS* and *isQValid*. AC5TC-Sparse only has to maintain *FS* and one integer, so $\Theta(r \cdot d)$ integers. The cost the algorithms have to pay during backtracking is proportional to the number of changes in the structures when trailing is used. Trailing is a technique to maintain information during the search, recording the modifications to the trailed structures and undoing them on backtracking. The nature, integer or Boolean, of the elements of the structure doesn't matter: a cost of $O(1)$ has to be paid to store/restore them. Each change is recorded in the trailing record of the algorithms. We thus compare here the number of modifications to their structures, which is the size of their trailing record. For one table constraint, along a branch in the search tree containing m nodes, if k tuples of the table are found to be Q-invalid, the size of the trailing record due to *FS* is $O(r \cdot k)$. In addition, AC5TC-Bool has k elements in its trailing record due to *isQValid* and AC5TC-Sparse has at most m elements in its record, because only *size* is trailed. If at most one tuple is found Q-invalid in each node, the records have the same size, but usually we have $m < k$. Also, the contribution to the record size due to the Q-validity structure seems small compared to the contribution of *FS*. However the upper bound on the number of modifications to *FS* will only be attained if each removed tuple updates the first support of $r - 1$ literals. In practice, this is a rare

event. The choice of the Q-validity structure thus has a large impact on the performance of the propagators, as we will see in the experimental results of the algorithms.

4.3 A VARIATION BASED ON RECOMPUTATION

Using the Q-validity of the tuples while traversing the *next* structure has a drawback. A literal that is no longer domain consistent may have its first support updated several times before being removed. In a worst case scenario, a literal with a lot of Q-valid (but not valid) supports may have its first support updated to each of the Q-valid supports before being removed. Example 6 illustrates this problem.

Example 6. In a binary table constraint c where $scope(c) = \{x, y\}$, let's suppose a literal (y, b) has only 3 supports (ordered as they are in the table): τ_1, τ_2 and τ_3. Let's suppose that $\tau_1[x] = a$, $\tau_2[x] = b$ and $\tau_3[x] = c$. If the propagation queue contains (x, a), (x, b) and (x, c) and they are popped out in that order, the first support for (y, b) (initially τ_1) will be updated to τ_2, then to τ_3, and finally to $\top$.

The solution to this problem is to work directly with the validity of the tuples instead of using the Q-validity. In the previous scenario, the literal would have been removed the first time a new valid support had been searched for. We thus propose a variation of the AC5TC algorithm, called AC5TC-Recomp, that works with the validity of the tuples. AC5TC-Recomp does not require any data structure to store Q-validity information. Rather, the validity of the tuples is tested when needed and not stored. Even with this switch from Q-validity to validity, AC5TC-Recomp still uses the same *next* structure as used by AC5TC. However, the *FS* structure has to be slightly changed together with its invariant. The new structure is called *FUS* and stores, for each literal, the index of the first useful support in the table. Its new invariant can be found below and is satisfied before dequeuing an element of Q.

$$\forall x \in scope(c), \forall a \in D(x) :$$
$$FUS[x, a] = i \Leftrightarrow [\tau_{c,i}[x] = a \wedge i \neq \top \wedge \tau_{c,i} \in D(scope(c), Q, c) \wedge (\forall j < i : \tau_{c,j}[x] = a \Rightarrow \tau_{c,j} \notin D(scope(c)))]$$
$$\forall x, y \in scope(c), \forall a \in D(x), b \in D(y, Q, c) \backslash D(y) :$$
$$(FUS[x, a] = i \wedge \tau_{c,i}[y] = b) \Rightarrow FUS[y, b] \leq FUS[x, a]$$

The first part of the invariant states that for a literal in the domain, the first useful support is Q-valid and different from $\top$. It also states that all the possible supports before the first useful support are invalid. This is the first difference from the invariant for *FS* in AC5TC. Recall that in the *FS* invariant, the preceding candidate supports were Q-invalid. This difference arises because AC5TC-Recomp is not using Q-validity information. When updating the first useful support of a tuple, this can avoid updating the first useful support to Q-valid tuples that are invalid. The second part of the invariant is dedicated to ensuring that the update process will not miss a *FUS* update for some literal. This will be explained in more detail later, together with the update process.

The `post` method of AC5TC-Recomp is very similar to the `postTC` method of Algorithm 2. It also initializes the *next* data structure and *FUS* is initialized the same way *FS* is in `postTC`. No Q-validity is used in either of the `post` methods, and *FS* and *FUS* are exactly the same after the initialization. The only difference between `postTC` and the `post` method of AC5TC-Recomp is that AC5TC-Recomp does not call `initSpecStructTC` or `setQInvalidTC`.

The `valRemoveTC-Recomp` method is given in Algorithm 7. This has similarities with `valRemoveTC` (Algorithm 3). Indeed, when called for a constraint c, a variable y and a value b, `valRemoveTC-Recomp` cycles through the tuples $\tau_{c,i}$ where $\tau_{c,i}[y] = b$. It starts at $FUS[y,b]$ because the tuples before $FUS[y,b]$ are invalid and cannot be the first useful support for any valid literal. This is granted by the *FUS* invariant. For each of these tuples $\tau_{c,i}$, a new support is sought for each literal (x,a) for which $FUS[x,a] = i$. If no new support is found, `valRemoveTC-Recomp` includes (x,a) in $\triangle$, as was done in `valRemoveTC`. However, there are significant differences between `valRemoveTC-Recomp` and `valRemoveTC`. The first is the use of `seekNextSupportTC-Recomp` to update *FUS*. This method searches for a new valid support, traversing the *next* chains, and does not search for a Q-valid one. Recall that `seekNextSupportTC` (Algorithm 4) updates *FS* searching for the next Q-valid support. Another difference between `valRemoveTC-Recomp` and `valRemoveTC` is the test $\tau_{c,i}[x] \in D(x)$ in line 8 of `valRemoveTC-Recomp`. This test is the equivalent of the one in line 13 of Algorithm 3. This test is needed here at line 8 to maintain the second part of the *FUS* invariant. This part of the invariant guarantees that for each literal in the queue, its first useful support is before the first useful support of the valid literals in this tuple. This allows `valRemoveTC-Recomp`(c,y,b) to look for updates to make from $FUS[y,b]$ towards the end of the table. In order to see the need for the second part of the invariant, let's suppose it were not present and the test $\tau_{c,i}[x] \in D(x)$ were pushed inside line 12 (as it is

in Algorithm 3). Let's suppose we are executing `valRemoveTC-Recomp` for the literal (y,b). Let's also suppose that the first useful support of a literal (x,a) in the propagation queue contains (y,b). Without the test $\tau_{c,i}[x] \in D(x)$ of line 8, a new valid support for (x,a) would be sought to update $FUS[x,a]$. This would automatically set $FUS[x,a]$ to $\top$ because all supports of (x,a) contain (x,a) and are thus invalid. Setting $FUS[x,a]$ to $\top$ would prevent the call to `valRemoveTC-Recomp` for (x,a) to update FUS for the literals that have a tuple with (x,a) as first useful support. This may lead the algorithm to leave non-GAC literals out of $\triangle$. This is already implicitly present in the FS invariant (Section 4.2), thanks to the Q-validity of the first support granted for the valid literals and the literals in the queue and the Q-invalidity of the candidate supports before. This guarantees that the first support of each literal in the queue is before the first support of the literals in this tuple.

```
1  valRemoveTC-Recomp(in c: Constraint;in y: Variable;
2                     in b: Value; out △: Set of Values){
3  // Pre: c ∈ C, c is a table constraint and b ∉ D(y,Q,c)
4  // Post: Inc(c,D(X,Q,c)) ∩ Cons(c,y,b) ⊆ △ ⊆ Inc(c) ∩ Cons(c,y,b)
5      △ = ∅;
6      i = c.FUS[y,b];
7      while(i!=⊤){
8          forall(x in scope(c): x!=y && τ_{c,i}[x] ∈ D(x)){
9              a = τ_{c,i}[x];
10             if (c.FUS[x,a]==i){
11                 c.FUS[x,a]=seekNextSupportTC-Recomp(c,x,i);
12                 if(c.FUS[x,a]==⊤)   △ += (x,a);
13             }
14         }
15         i = c.next[y,i];
16     }
17 }
```

Algorithm 7: Method `valRemoveTC-Recomp` for Table Constraints.

From a complexity point of view, it may seem inefficient to test the validity of the tuples when searching for a new support. The test $\tau_{c,i} \in D(scope(c))$ is $O(r)$. However, using the validity has an advantage: it allows `valRemoveTC` to output a larger $\triangle$ set than the previous algorithms. The validity information is stronger than the Q-validity information since each valid tuple is also Q-valid but not conversely. With this larger $\triangle$, the domains at the fixed point will be the same for all algorithms (since all compute domain consistency) but the method `valRemoveTC-Recomp` of AC5TC-Recomp might be called less often during the GAC fixed point.

```
1  function seekNextSupportTC-Recomp(in c: Constraint;
2                  in x: Variable; in i: Index)  : Index {
3  // Pre: c ∈ C, c is a table constraint, x ∈ scope(c), 1 ≤ i ≤ c.length
4  // Post: return the first index j greater than i of a valid tuple
5  //         with τ_{c,j}[x] == τ_{c,i}[x]
6      i = c.next[x,i];
7      while(i!=⊤){
8          if(τ_{c,i} ∈ D(scope(c)))   return i;
9          i = c.next[x,i];
10     }
11     return ⊤;
12 }
```

Algorithm 8: The `seekNextSupportTC-Recomp` Function of AC5TC-Recomp.

Proposition 2. *AC5TC-Recomp is correct. It has a time complexity of* $O(r^2 \cdot t + r \cdot d)$ *and a space complexity of* $O(r \cdot t + r \cdot d)$ *per table constraint along a branch in the search tree.*

Proof. Both methods `postTC-Recomp` and `valRemoveTC-Recomp` satisfy their specifications (Specification 1). Hence, AC5TC-Recomp is correct. The `post` method of AC5TC-Recomp has the same complexity as that of AC5TC: $O(r \cdot t + r \cdot d)$. The complexity of all the executions of `valRemoveTC-Recomp` is $O(r^2 \cdot t)$ per table constraint. The justification of this complexity is similar to that for `valRemoveTC` (Proposition 1). For a variable y, all the executions of `valRemoveTC-Recomp` follow the chains in the *next* structure for each different value of the variable. The total number of elements in those chains is t since the tuples have one value per variable. With r variables, lines 8 to 15 are executed $O(r \cdot t)$ times. Without line 11, those lines have a complexity of $O(r)$. Without line 11, the complexity of all the executions of `valRemoveTC-Recomp` is $O(r^2 \cdot t)$. For a particular variable, the loop in line 6 of `seekNextSupportTC-Recomp` can only be executed $O(t)$ times in all during the entire fixed point (totalled up for all its values). Indeed, the next chains are traversed at most once. The test $\tau_{c,i} \in D(scope(c))$ in line 8 of `seekNextSupportTC-Recomp` being $O(r)$, the executions of line 11 of `valRemoveTC-Recomp` are $O(r^2 \cdot t)$ in all during the entire fixed point. Hence the complexity of $O(r^2 \cdot t)$ for `valRemoveTC-Recomp` during a fixed point. □

The spatial complexity of AC5TC-Recomp is $\Theta(r \cdot t + r \cdot d)$, as it uses the same *next* structure as used in the previous section and *FUS* is $\Theta(r \cdot d)$. AC5TC-Recomp has to maintain *FUS*, which means it has to maintain $\Theta(r \cdot d)$

integers. When the GAC fixed point is reached, the *FUS* structure is the same as the *FS* structure in the previous algorithms. The number of modifications to the structure that AC5TC-Recomp has to store in its trailing record for *FUS* is thus the same as the number of modifications the previous algorithms had to store for *FS*. In addition, along a branch in the search tree containing m nodes, if k tuples are found Q-invalid, AC5TC-Bool has k elements in its trailing record while AC5TC-Sparse has at most only m integers in its record. AC5TC-Recomp doesn't have to maintain any data structure besides *FUS*.

Despite being non-optimal, in the experiments AC5TC-Recomp improves on state-of-the-art algorithms for some classes of problems. It is the (previously unpublished) table constraint propagator implemented in the Comet system.

4.4 OPTIMAL GAC PROPAGATORS

In all of the previously presented algorithms, the successive updates of their support structures may revisit the same tuple several times. This comes from the static nature of the *next* structure. Indeed, there is no guarantee that a tuple visited in a particular *next* chain, and found to be Q-invalid or invalid, will not be revisited later in another chain. This redundant work has a cost. This cost is particularly penalizing for AC5TC (Section 4.2). In method `valRemoveTC` (Algorithm 3), all the executions of the `seekNextSupportTC` (line 12 of Algorithm 3) take $O(r \cdot t)$, assuming `isQValidTC` takes constant time. However, the total time complexity of all the executions of `valRemoveTC` is $O(r^2 \cdot t)$. This makes AC5TC reach the non-optimal $O(r^2 \cdot t + r \cdot d)$ time complexity per table constraint. Recall that the optimal time complexity per table constraint is $O(r \cdot t + r \cdot d)$. To remedy this situation, the idea is to use a dynamic structure to store the supports for each literal instead of the static *next* structure. Dynamic structures are maintained during the search. We call these the *collections* of supports. These dynamic structures thus keep the Q-valid supports for the values in $D(X, Q, c)$. This avoids revisiting tuples, because as soon as a tuple is detected to be Q-invalid, it is removed from the active collections it belongs to. Those collections are stored in a single structure, called *Col*. More formally, for a table constraint c, the structure containing the collections, *Col*, satisfies the following invariant before dequeuing an element of Q:

$$\forall x \in Vars(c), \forall a \in D(x, Q, c) :$$
$$Col[x, a] = \{1 \leq i \leq c.length \mid \tau_{c,i}[x] = a \wedge \tau_{c,i} \in D(Vars(c), Q, c)\}$$

This invariant simply states that for each variable x and each value in its extended domain $D(x, Q, c)$, its collection contains all its Q-valid supports. In

order to allow different implementations for this collection structure, we have designed a generic propagator. This propagator uses the abstract methods specified in Specification 2. Those methods simply form iterators over *Col*. They enclose all interactions with the collections in *Col*. For the iterator requirement, we assume that no remove operation is performed on a collection when iterating over it. We will refer to them as the specific functions as they are specific to the concrete propagators implementing the generic propagator.

```
1  initSpecStruct(in c: Constraint,
2                 in Col_init: Matrix of Arrays)
3  //pre: the collections in Col_init are sorted
4  //initializes specific structures implementing Col to the
5  //initial collections contained in the matrix Col_init
6  //in Col_init, a collection is represented by an array.
7
8  firstInCollection(in c: Constraint;
9                 in x: Variable; in a: Value): Index
10 //returns the first element in Col[x,a]
11
12 nextInCollection(in c: Constraint;
13                in x: Variable; in i: Index): Index
14 //returns the element following i in Col[x,a] where a = τ_{c,i}[x]
15
16 //functions firstInCollection, nextInCollection and the test
17 //nextInCollection≠ ⊤ form iterators over the collections
18 //in Col
19
20 isCollectionEmpty(in c: Constraint;
21                in x: Variable; in a: Value): Boolean
22 //returns true iff Col[x,a] is empty
23
24 removeFromCollection(in c: Constraint,in x: Variable,
25                      in i: Index)
26 //removes i from Col[x,a] where a = τ_{c,i}[x]
```

Specification 2: The abstract methods used by the generic optimal table constraint propagator

The optimal generic algorithm for table constraints is called AC5TCOpt. The `post` and `valRemove` methods for AC5TCOpt are presented in Algorithms 9 and 10. Method `postTCOpt` first computes the initial collection Col_{init}. Col_{init} is represented by a matrix containing one array per literal. For a literal, its array in Col_{init} contains the indexes of its supporting tuples. `post-`

`TCOpt` then initializes the specific structures of the propagator by calling `initSpecStruct`. It then removes all the values with no valid support. The method `valRemoveTCOpt`(c,y,b) uses methods `firstInCollection`(c,y,b) (line 6), `nextInCollection`(c,y,i) (line 15), and the test $i \neq \top$ (line 7) to traverse the collection for (y,b). These are the only tuples that become Q-invalid when (y,b) is popped out of Q. These tuples are removed from the collections they belong to and the literals left without any Q-valid support are included in $\triangle$.

```
1  postTCOpt(in c: Constraint; out △: Set of Values){
2  // Pre: c ∈ C, c is a table constraint
3  // Post: △ = Inc(c) + initialization of the data structures
4      △ = ∅;
5      forall(x in scope(c),a in D(x)) Col_init[x,a]=[];
6      forall(i in 1..c.length: τ_{c,i} in D(scope(c)))
7          forall(x in scope(c))
8              Col_init[x,τ_{c,i}[x]].append(i);
9      initSpecStruct(c,Col_init);
10     forall(x in scope(c),a in D(x))
11         if(isCollectionEmpty(c,x,a)) △ += (x,a);
12 }
```

Algorithm 9: The post method of optimal AC5TCOpt table constraint propagator

Proposition 3. *Assuming that* `initSpecStruct`, `firstInCollection`, `nextInCollection`, `isCollectionEmpty` *and* `removeFromCollection` *are correct, AC5TCOpt is correct. Assuming a complexity of* $O(r \cdot t + r \cdot d)$ *for* `initSpecStruct`, *and* $O(1)$ *for the other specific functions, the time complexity of AC5TCOpt is the optimal* $O(r \cdot t + r \cdot d)$ *per table constraint along a branch in the search tree.*

Proof. After the execution of Method `postTCOpt`, the *Col* invariant is respected. Let's suppose that the *Col* invariant is true before dequeuing a literal (y,b) from Q. As the invariant is verified before the execution, the only tuples that need attention are the tuples that become Q-invalid. Since the specific methods are correct, `valRemoveTCOpt` iterates through all the previously Q-valid tuples $\tau_{c,i}$ where $\tau_{c,i}[y] = b$. Those tuple indexes are removed from all the sets they belong to. The *Col* invariant is thus restored. With the *Col* invariant, it is straightforward to prove that AC5TCOpt is correct. Indeed, both `postTCOpt` and `valRemoveTCOpt` meet Specification 1 since the $\triangle$s are filled with the literals for which the collection in *Col* is empty. With an

```
1  valRemoveTCOpt(in c: Constraint; in y: Variable;
2                 in b: Value; out △: Set of Values) {
3  // Pre: c ∈ C, c is a table constraint and b ∉ D(y,Q,c)
4  // Post: △ = Inc(c,D(X,Q,c)) ∩ Cons(c,y,b)
5    △ = ∅;
6    i = firstInCollection(c,y,b);
7    while(i ≠ ⊤){
8      forall(x in scope(c) : x ≠ y){
9          removeFromCollection(c,x,i);
10         a = τ_{c,i}[x];
11         if(isCollectionEmpty(c,x,a) && a in D(x)){
12             △ += (x,a);
13         }
14     }
15     i = nextInCollection(c,y,i);
16   }
17 }
```

Algorithm 10: The valRemove method of optimal AC5TCOpt table constraint propagator

$O(r \cdot t + r \cdot d)$ complexity for `initSpecStruct` and an $O(1)$ complexity for `isCollectionEmpty`, postTCOpt is obviously $O(r \cdot t + r \cdot d)$. The complexity of `valremoveTCOpt` is $O(r \cdot t)$. Indeed, each execution of lines 6 to 16 leads to different values for i in $\{1, \ldots, t\}$. This is due to the fact that the indexes of the visited tuples are removed from all the collections they belong to. Lines 8 to 15 are thus executed at most t times along a branch in the search tree. The complexity of those lines being $O(r)$, the overall complexity of `valRemove` is $O(r \cdot t)$ per table constraint (assuming the presence of other constraints on which domain consistency is also applied). □

We now present two different implementations of AC5TCOpt. They differ in the data structure they use to backup the implementation of the functions `firstInCollection`, `nextInCollection`, `isCollectionEmpty` and `removeFromCollection`.

4.4.1 *AC5TCOpt-Tr*

The first propagator reuses an idea similar to the *next* structure of AC5TC (Section 4.2) and AC5TC-Recomp (Section 4.3). In order to implement the specific functions of AC5TCOpt, iterating through the collections in *Col*, this

new structure has to be dynamic and it has to allow the removal of tuples in the collection in $O(1)$. Indeed, the collections in *Col* depend on the domains of the variables, and thus have to change during the search. This is why we used two arrays to represent the index collections in *Col*: *nextTr* and *predTr*. They can be viewed as a double-linked list version of the single-linked list *nextTr*. The first structure, *nextTr*, contains, for each index i and each variable x, the next tuple index in $Col[x, \tau_{c,i}[x]]$. The second one, *predTr*, contains the preceding tuple index in $Col[x, \tau_{c,i}[x]]$. This propagator also uses an array, called *FS*, referring, for each literal (x, a), to the first tuple index in $Col[x, a]$. The name *FS* has been chosen for the similarities with the *FS* array in AC5TC (Section 4.2). The *nextTr*, *predTr* and *FS* data structures should be maintained during the search, since *Col* depends on the current domains. For a variable x and a value $a \in D(x, Q, c)$, $Col[x, a]$ can be enumerated by following the *nextTr* chain, starting at $FS[x, a]$. The order between the tuples in *nextTr* and *predTr* is fixed to be that of the table order. The collections in *Col* are thus here ordered with respect to this order. As the *nextTr* and *predTr* are trailed through the execution of the propagator, we call the AC5Opt algorithm using those structures AC5TCOpt-Tr (for AC5 Optimal Table Constraint Propagator with Trailing).

The implementation of the specific functions (Specification 2) of AC5TC-Opt-Tr is presented in Algorithm 11. Method `initSpecStruct` initializes *nextTr*, *predTr* and *FS*. As explained before, *FS* contains the beginning of the collection for each different literal, in this case, the smallest index (the arrays in Col_{init} are sorted). The functions `firstInCollection` and `nextInCollection` are straightforward. For `isCollectionEmpty`, if the smallest index of an element in a collection is $\top$, that means that the collection for the literal is empty. The `removeFromCollection` also takes into account the increasing order of the indexes in the structures. It has to distinguish two cases. In the first one, the index to remove is the first one (line 34). In this case, the first one is set to be the index following it. This effectively removes index i from the collection because the traversal of the collection is performed from *FS* to the end of the table. The second case (line 36) is the one where the index to remove is after the first one. In this case, it must be removed from the *nextTr* and *predTr* structures to remove it from the collection. The index i is never smaller than $FS[x, a]$ because that would mean that `removeFromCollection` would have been called to remove i from x's collection but not from y's.

Proposition 4. *The implementation of* `initSpecStruct`, `firstInCollection`, `nextInCollection`, `isCollectionEmpty` *and* `removeFromCollection` *is correct. The time complexity of* `initSpecStruct`

```
1  initSpecStruct(in c: Constraint,
2                 in Col_init: Matrix of Arrays){
3     forall(x in scope(c), i in 1..c.length)
4        c.nextTr[x,i] = ⊤; c.predTr[x,i] = ⊥;
5     forall(x in scope(c), a in D(x)){
6        nSup=len(Col_init[x,a]);
7        if(nSup==0)
8           c.FS[x,a]=⊤;
9        else
10          c.FS[x,a]=Col_init[x,a][1];
11          forall(j in 1..nSup-1){
12            c.nextTr[x,Col_init[x,a][j]]=Col_init[x,a][j+1];
13            c.predTr[x,Col_init[x,a][j+1]]=Col_init[x,a][j];
14          }
15    }
16 }
17
18 firstInCollection(in c: Constraint; in x: Variable;
19                   in a: Value):Index{
20     return c.FS[x,a];
21 }
22
23 nextInCollection(in c:Constraint; in x: Variable;
24                   in i: Index):Index{
25     return c.nextTr[x,i];
26 }
27
28 isCollectionEmpty(in c: Constraint; in x: Variable;
29                   in a: Value):Boolean{
30     return c.FS[x,a] == ⊤;
31 }
32
33 removeFromCollection(in c: Constraint;
34                   in x: Variable, in i: Index){
35    a = τ_{c,i}[x]
36    if(c.FS[x,a] == i){
37      c.FS[x,a] = c.nextTr[x,i];
38    } else{    // i > FS[x,a]
39      if (c.predTr[x,i]!=⊥)
40        c.nextTr[x,c.predTr[x,i],c] = c.nextTr[x,i,c];
41      if (c.nextTr[x,i]!=⊤)
42        c.predTr[x,c.nextTr[x,i],c] = c.predTr[x,i,c];
43    }
44 }
```

Algorithm 11: Specific Methods of AC5TCOpt-Tr

is $O(r \cdot t + r \cdot d)$. All the other methods are $O(1)$. The time complexity of AC5TCOpt-Tr is the optimal $O(r \cdot t + r \cdot d)$ per table constraint along a branch in the search tree.

Proof. The proof that each function respects its individual specification (Specification 2) is straightforward. The last part of the specification to prove is that the functions `firstInCollection`, `nextInCollection` and the test `nextInCollection`$\neq \top$ form iterators over the collections in *Col*. This is due to the fact that *FS* always referrs to the first element in its collection and that the only elements removed from the *nextTr* and *predTr* chains are the ones removed from the collections with `removeFromCollection`. The time complexity of `initSpecStruct` is $O(r \cdot t + r \cdot d)$ because the total number of supports for a variable is $O(t)$, hence is the number of elements in the *Col* structures for a variable. This guarantees that the loop in line 10 is $O(t)$ for all the values of a variable. The loop in line 4 is thus $O(r \cdot t + r \cdot d)$. AC5TCOpt-Tr is then $O(r \cdot t + r \cdot d)$. □

AC5TCOpt-Tr has a spatial complexity of $\Theta(r \cdot t + r \cdot d)$ since *nextTr* and *predTr* are $\Theta(r \cdot t)$ and *FS* is $\Theta(r \cdot d)$. All these structures have to be trailed during the search. AC5TC-Tr has thus $\Theta(r \cdot t + r \cdot d)$ integers to backtrack. Compared to the previous algorithms, the difference in backtrackable structure is *nextTr*/*predTr*. For a table constraint, along a branch in the search tree of length m, if k of its tuples are found Q-invalid, AC5TCOpt-Tr has $\Theta(r \cdot k)$ integers in its trailing record. Indeed, upon finding a Q-invalid tuple, for the r variables, either FS is modified or the two pointers *nextTr*/*predTr*. This is much more than the previous algorithms. Along the same branch, they have $O(r \cdot k)$ integers to trail for *FS* and an additional k elements for AC5TC-Bool and at most m integers for AC5TC-Sparse for the Q-validity structures. AC5TC-Recomp only has to backtrack *FS*. As explained before, the worst case scenario leading to $O(r \cdot k)$ integers in the record for *FS* is not frequent.

4.4.2 *AC5TCOpt-Sparse*

The implementations of the specific methods of AC5TCOpt presented in the previous section have one drawback: AC5TCOpt-Tr has to maintain $\Theta(r \cdot t + r \cdot d)$ integers during the search. Trailing such a large number of integers can be costly. By changing the data structures used to support the implementation of the specific AC5TCOpt methods, we can obtain a propagator trailing only $\Theta(r \cdot d)$ integers. This new implementation, called AC5TCOpt-Sparse, is the topic of this section.

The fundamental change to AC5TCOpt-Tr is to not maintain the order of the elements in the collections. Indeed, inside the *nextTr* chains, the elements are ordered with respect to their order of appearance in the table. This is an unnecessary requirement. All that is needed is that those elements match the ones in the collections. We can therefore replace the *nextTr* and *predTr* structures with efficiently backtrackable sets. The structures we chose for representing those sets are an adapted version of sparse sets, introduced in Section 4.2.

Recall that a sparse set structure uses two different arrays for representing a set: *Map* and *Dyn*. In addition, Sparse Sets also maintain the size (*size*) of the set. *Map* contains, for each entry of the set, its position in the array *Dyn*. *Dyn* contains the elements of the set before the index *size*, and the removed ones after *size*. There will be one such set per collection in *Col*, thus per literal. Each set will contain tuple indexes from *Col*. For a variable x and a tuple index i, $\tau_{c,i}[x]$ is the only value for x. The index i can thus only be in one collection from *Col*, thus in one sparse set for x. This means that one Map array can be shared by all the literals of the same variable. Indeed, for a tuple i and a variable x, $Map[i]$ will represent the position of i in the *Dyn* array of the literal $(x, \tau_{c,i}[x])$. The link between the arrays *Map* and *Dyn* can be formalized as

$$\forall x \in X, \quad \forall i,j : 1 \leq i,j \leq t : \\ Map[x][i] = j \Leftrightarrow Dyn[x, \tau_{c,i}[x]][j] = i$$

We will refer to this as the *Dyn*/*Map* invariant. Sharing the *Map* between all the values of a variable allows the size of the *Map* array to be $\Theta(t)$ (for each variable). After the creation of the structures, previously unseen elements are never added to *Col*. This allows the size of the *Dyn* array to be fixed to the initial number of support of its literal. Figure 4.5 shows an example of the structures for variable x. In this example, the arrays $Dyn[x,a]$ and $Dyn[x,b]$ can be used to traverse the collections. The array $Map[x]$ can be used, together with the table of the constraint, to locate an element in the different *Dyn* arrays in $O(1)$. Amongst these structures, only *size* has to be trailed.

The implementation of the specific methods (Specification 2) of AC5TCOpt for AC5TCOpt-Sparse is given in Algorithm 12. Method `initSpecStruct` fills the *Dyn*, *Map* and *size* structures. Method `firstInCollection` returns the first element in *Dyn* if the sparse set is not empty, $\top$ otherwise. As the elements are swapped in *Dyn* on removal, the first element is not always the one with the smallest index. Method `nextInCollection` returns the element following the current element in *Dyn* if it is at an index smaller than the *size* of this set. It returns $\top$ otherwise. Method `removeFromCollection` swaps the element that is to be removed with the last one of the right *Dyn* array

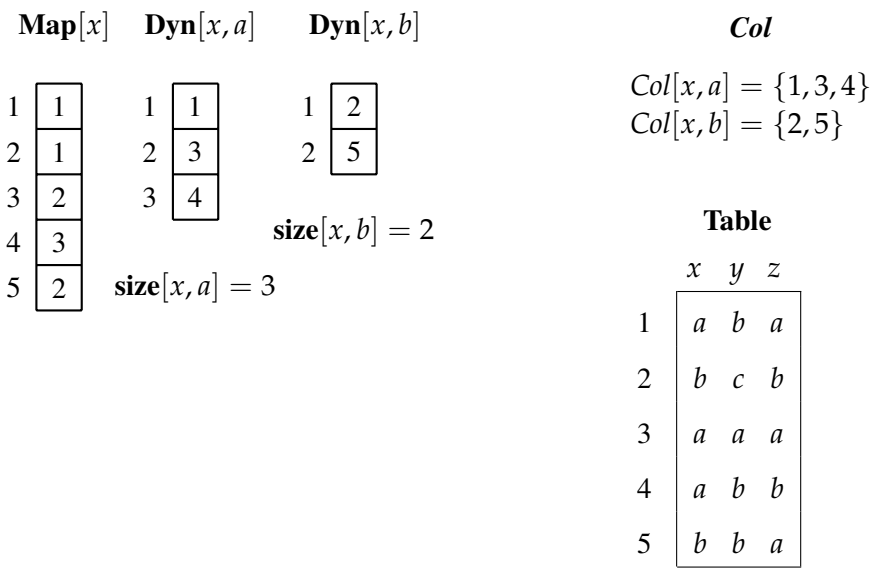


Figure 4.5: Example of the Sparse Set structures used by AC5TCOpt-Sparse for variable x

and updates the *Map* structure. This keeps the link between *Map* and *Dyn* consistent. This is the standard procedure for removing an element from a sparse set, adapted to the present case.

Proposition 5. *The implementation of* `initSpecStruct`, `firstInCollection`, `nextInCollection`, `isCollectionEmpty` *and* `removeFromCollection` *for AC5TCOpt-Sparse is correct. The time complexity of* `initSpecStruct` *is* $O(r \cdot t + r \cdot d)$. *All the other methods are* $O(1)$. *The time complexity of AC5TCOpt-Sparse is the optimal* $O(r \cdot t + r \cdot d)$ *per table constraint along a branch in the search tree.*

Proof. The correctness of the specific functions with respect to their individual specifications (Specification 2) is straightforward provided that the *Dyn* and *Map* arrays are consistent. They are consistent if and only if the *Dyn*/*Map* invariant is respected. The invariant is trivially respected after the `initSpectStruct`. The only function that changes *Dyn* and *Map* after their creation is `removeFromCollection`. If the invariant is respected before `removeFromCollection`, it is respected after. The invariant is therefore always respected. The functions `firstInCollection`, `nextInCollection` and the test `nextInCollection`$\neq \top$ thus form iterators over the collections in *Col*, the elements from $Col[x,a]$ being in $Dyn[x,a]$ between indexes 1 and

```
1  initSpecStruct(in c: Constraint,
2                 in Col_init: Matrix of Arrays){
3     forall(x in scope(c), a in D(x)){
4        j=1;
5        forall(i in Col_init[x,a]){
6           c.Dyn[x,a][j] = i;
7           c.Map[x][i] = j;
8           j+=1;
9        }
10       c.size[x,a] = Col_init[x,a].length;
11    }
12 }
13
14 firstInCollection(in c: Constraint;
15                 in x: Variable; in a: Value){
16    if(c.size[x,a] > 0)
17       return c.Dyn[x,a][1];
18    else
19       return ⊤;
20 }
21
22 nextInCollection(in c: Constraint;
23                 in x: Variable; in i: Index){
24    if(c.Map[x][i] < c.size[x,a]){
25       return c.Dyn[x,τ_{c,i}[x]][c.Map[x][i]+1];
26    }else{
27       return ⊤;
28    }
29 }
30
31 isCollectionEmpty(in c: Constraint;
32                 in x: Variable; in a: Value){
33    return c.size[x,a] == 0;
34 }
35
36 removeFromCollection(in c: Constraint; in x: Variable,
37                 in i: Index){
38    a = τ_{c,i}[x];
39    c.Dyn[x,a][c.Map[x][i]] = c.Dyn[x,a][c.size[x,a]];
40    c.Dyn[x,a][c.size[x,a]] = i;
41    c.Map[x][c.Dyn[x,a][c.Map[x][i]]] = c.Map[x][i];
42    c.Map[x][i] = c.size[x,a];
43    c.size[x,a] -= 1;
44 }
```

Algorithm 12: Specific Methods of AC5TCOpt-Sparse

$size[x,a]$. Recall that no remove operation is performed on a collection while iterating over it (hypothesis from *Col*). The $O(r \cdot t + r \cdot d)$ time complexity of `initSpecStruct` is provided by the $O(t)$ elements in the collections for all the values of a variable. □

Although the time complexity of AC5TCOpt-Sparse is the same as that of AC5TCOpt-Tr (Section 4.4.1), the advantage of AC5TCOpt-Sparse is the number of integers to backtrack. AC5TCOpt-Sparse backtracks only the *size* array, which contains one integer per literal. It has thus only $\Theta(r \cdot d)$ integers to backtrack, compared to the $\Theta(r \cdot t + r \cdot d)$ integers that AC5TCOpt-Tr has to trail. More precisely, for a table constraint, on a branch in the search tree, if k tuples are found Q-invalid, AC5TCOpt-Sparse has $O(r \cdot k)$ integers on its trailing record. This worse case scenario corresponds to the situation where each tuple detected Q-invalid updates the collections of r different literals. Otherwise, only one integer is stored on the record when multiple updates of the same collection is performed at a node. For a table constraint, along a branch in the search tree detecting k tuples from the table to be Q-invalid, AC5TCOpt-Tr has a total number of integers in its trailing record of $\Theta(r \cdot k)$. AC5TCOpt-Sparse always has a number of integers in its trailing record that is less than or equal to the number in AC5TCOpt-Tr record, but most of the time, it has strictly fewer integers in its trail.

The total spatial complexity of AC5TCOpt-Sparse is $\Theta(r \cdot t + r \cdot d)$. The size of *Dyn* is $\Theta(r \cdot t)$ since each tuple index is in exactly r Sparse Sets, the *Map* structure is $\Theta(r \cdot t)$ ($\Theta(t)$ for each variable), and the size structure is $\Theta(r \cdot d)$.

4.5 EXPERIMENTAL RESULTS

All proposed algorithms have been implemented on top of Comet. The core of the Comet system being closed source, a core implementation is not possible. For comparison, classical constraint-based algorithms have also been implemented on top of Comet. The AC3 and AC3rm algorithms [LH+07], designed for binary constraints, have also been reimplemented. In our implementation, these algorithms use a dichotomic search in the table to verify that a tuple is allowed by the constraint. The GAC3-Allowed algorithm has been chosen for the comparison because it is the standard GAC3 algorithm for non-binary table constraints [Lec09]. Three existing state-of-the-art methods were also reimplemented: The MDD^c algorithm from [CY10], the STR2+ algorithm from [Lec11] and STR3 from [LLY12]. For MDD^c, the order of the variables can have a big impact on the performance of the algorithm. Indeed, the order

strongly influences the size of the constructed MDD. Unfortunately, obtaining the perfect order is NP-Complete [CY10]. In the experiments, we used the variable order in the instances as the order in the MDDs. For the STR2+ reimplementation, the array *lastSize* was used. Some optimization can be obtained by reusing the structures constructed for a propagator between different constraints relying on the same table but with different scopes. This is the case, for instance, for the *next* structure of our propagators or the MDD of MDD^c. This optimization has not been used in our test for any of the propagators. All experiments were conducted on an Intel Xeon 2.53GHz using Comet 2.1.1. The algorithms are compared within a MAC search. The problems have been selected because they offer very different constraint arities. Some of them contain only binary tables while other contain up to arity 20 table constraints. This section thus presents results on the geometric problem, on the Langford problem, on the Traveling Salesman Problem, on the RandRegular problem, on fully random instances, on Crossword instances, and on modified Renault instances. All the instances used are available at `http://becool.info.ucl.ac.be/resources/positive-table-constraints-benchmarks`

For each instance set, the experimental results report the mean execution times in seconds (*totTime*), the mean "posting" times in seconds (*postTime*), the number of propagator calls (*nProp*), the percentage to the best with respect to execution time (*%best*), the mean of percentage to the best algorithm in terms of execution time (μ*%best*), the number of validity checks (*valChk*), Q-validity checks (*QvalChk*), and the number of pointers followed (*pFollow*). The difference between the *%best* and μ*%best* is the following: for *%best*, the execution times are averaged before computing the quantity. There is thus one best algorithm. For μ*%best*, the percentages are computed instance by instance and aggregated with a geometric mean at the end. This measure takes into account that different instances may have different best algorithms. The μ*%best* measure uses a geometric mean as suggested in [FW86]. The last reported quantity, *pFollow*, represents different quantities for different algorithms. For GAC3-Allowed, it corresponds to the number of times the tuples are accessed. For the AC5TC algorithms and AC5TCOpt-Tr, it is defined as the number of times the *next* or *nextTr* structures are used to traverse the table. For MDD^c, it corresponds to the number of edges followed in the MDD structure. Although referring to different quantities, *pFollow* is useful for comparing the behavior of the propagators as it reflects the usage of their specific structures. For each instance set, scatter plots of AC5TCOpt-Sparse versus STR2+ and STR3 are given. In those scatter plots, each point is an instance. The x coordinate of a point is the time taken by the algorithm on the bottom of the plot and its y coordinate, the time taken by the other algorithm. A point with coordinates

(5,10) means that this instance has been solved in 5 seconds by the algorithm on the bottom and 10 by the other. The $x = y$ line is also displayed. The more points an algorithm has on its opposing side of the line $x = y$, the faster it is than the other. Those plots allow a detailed view, instance by instance, of the performance of the algorithms. STR2+, STR3 and AC5TCOpt-Sparse have been chosen because AC5TCOpt-Sparse is the fastest of our algorithms and STR2+, STR3 are its best competitors. For the binary benchmarks, scatter plots of AC5TCOpt-Sparse versus AC3rm are also given.

The search strategy is given for each benchmark. We used the terminology defined in [Bee06]. The *dom* variable heuristic first chooses the variable with the smallest domain. The *dom/deg* heuristic first chooses the variable with the smallest ratio domain size—the degree of the variable (the number of constraints in which it is involved). The *lexicographic* value ordering consists in trying first the smallest value with respect to the lexicographic ordering.

The Geometric Problem Instances of the geometric problem are random instances generated following a specific structure proposed by Rick Wallace [Wal05]. Each variable is randomly placed in the unit square. A fixed distance (less than $\sqrt{2}$) is randomly chosen. For each pair of variables (x, y), if the distance between their associated points is less than or equal to this fixed distance, the arc (x, y) is added to the constraint graph. Constraint relations are then created as they are in fully random CSP instances [XBHL07]. The constraints of this problem are thus binary. We use the instance set from [Lec], which has 100 instances. In this instance set, all the variables have the same domain, of size 20. The search strategy uses the heuristic *dom/deg* with lexicographic value ordering. A timeout of 5 minutes has been used. The quantity *%solv* gives the percentage of instances solved.

Table 4.1 presents the experimental results on geom instances. Figure 4.6 plots the percent best quantities for this instance set. All the quantities (except %*solv*) are computed on instances for which none of the techniques timeouts. All our propagators are faster than the state-of-the-art STR2+, STR3 and MDD^c. AC5TCOpt-Tr, AC5Opt-Sparse and AC5TC-Recomp are also better than the classical AC3, and GAC3-Allowed propagators. AC3rm is clearly the fastest strategy on those instances. It is also the best on each instance, as its μ%*best* is 100. AC5TCOpt-Sparse is the fastest of our propagators on those instances. Its performance is competitive with AC3rm. It is however not the best of our propagators on each instance. The instances on which it is beaten are the smallest, where AC5TC-Recomp is the best of our propagators. AC5TCOpt-Sparse is significantly faster than AC5TCOpt-Tr, due to the cost that AC5TCOpt-Tr has to pay to trail its structures. Checking the va-

propagator	totTime	postTime	nProp	%best	μ%best	%solv	valChk	QvalChk	pFollow
GAC3-Allowed	10.1	0.3	288 k	276	283	86	28 k	0	28 k
AC5TC-Bool	12.5	0.3	867 k	341	328	84	300	25 k	50 k
AC5TC-Sparse	10.8	0.2	867 k	295	271	86	300	25 k	50 k
AC5TC-Recomp	7.9	0.2	831 k	216	206	87	6 k	0	29 k
AC5TCOpt-Tr	9.6	0.8	867 k	263	412	87	300	0	13 k
AC5TCOpt-Sparse	6.5	0.4	867 k	178	236	87	300	0	0
MDD^c	14.7	1.6	288 k	401	694	86	0	0	65 k
STR2+	24.9	0.3	288 k	680	650	82	26 k	0	0
STR3	15.2	0.6	867 k	413	477	84	300	0	0
AC3	10.4	0.1	288 k	283	234	85	0	0	0
AC3rm	**3.7**	0.1	288 k	100	100	89	0	0	0

Table 4.1: Results of the propagators on the geom instances (times in seconds)

lidity (instead of the Q-validity) allows AC5TC-Recomp to follow less pointers than AC5TC-Bool and AC5TC-Sparse by performing longer jumps in the table. Moreover, as the tables are binary, the cost of a validity check is low. AC5TCOpt-Tr follows much fewer pointers than AC5TC-Bool and AC5TC-Sparse because it does not follow pointers to a previously inspected tuple.

Scatter plots for the geom instance set are given in Figure 4.7. In those scatter plots, next to each technique name, are the number of instances that are solved by this algorithm that triggered a timeout with the other algorithm. For instance, AC5TCOpt-Sparse solved five instances that caused STR2+ to timeout. Those instances are not in the plot. As we can observe in these scatter plots, AC3rm is faster than AC5TCOpt-Sparse on all instances except for the easiest ones. AC3rm solves two instances that caused AC5TCOpt-Sparse to timeout. AC5TCOpt-Sparse solves more instances and is faster than STR2+ and STR3 on all the instances except the easiest ones. We can also observe that STR3 is faster than STR2+. The time performances of those algorithms are proportional on this instance set.

Langford Number Problem The Langford number problem $L(k,n)$ is the problem of arranging k sets of numbers 1 to n into a sequence of numbers such that each occurrence of a number m is m numbers apart from its pre-

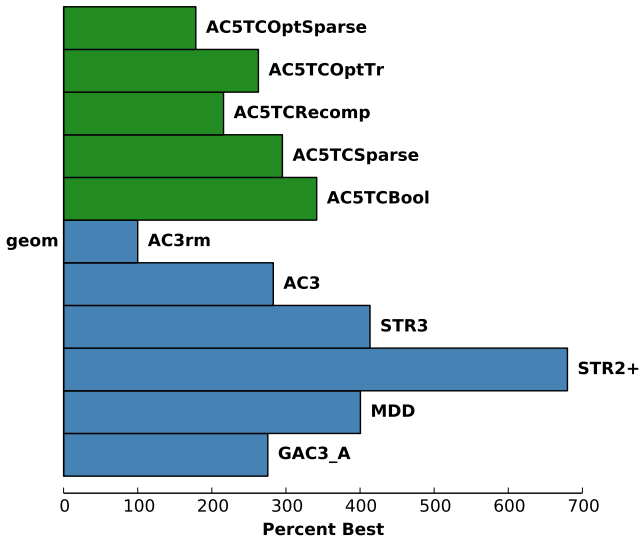


Figure 4.6: Percent best quantities for the geom instance set

vious occurrence. This is problem 24 of CSPLIB [3]. These problems can be fully modeled with binary (positive) table constraints. The instances with table constraints can be found in [Lec]. The search strategy used was *dom/deg* with lexicographic value ordering. Problems where all the propagators take more than 5 minutes have been removed from the sets. For $k = 2$, 12 instances are used: $n \in \{5..12, 15, 16, 19, 20\}$, for $k = 3$, 8 instances: $n \in \{3..10\}$ and for $k = 4$, 9 instances: $n \in \{3..11\}$. The results for k equal to 2, 3 and 4 can be found in Table 4.2. Figures 4.8, 4.9 and 4.10 plot the percent best quantities for $k = 2$, $k = 3$ and $k = 4$ respectively.

The fastest propagator on those instances is clearly AC3rm. However, except for AC5TC-Bool on the $k = 4$ set of instances, all our propagators are faster than the state-of-the-art STR2+, STR3 and MDD^c. AC5TCOpt-Sparse, AC5TCOpt-Tr and AC5TC-Recomp are faster than AC3 on the $k = 2$ set. AC5TCOpt-Sparse and AC5TC-Recomp are faster than AC3 on the $k = 3$ set. The three fastest of our propagators on those instances are AC5TCOpt-Sparse, AC5TCOpt-Tr and AC5TC-Recomp. They are also better than the classical GAC3-Allowed. Among our propagators, AC5TCOpt-Sparse is the fastest on those three instance sets. Our optimal AC5TCOpt-Tr is only faster than AC5TC-Recomp on the $k = 2$ instance set. Observe that the number of

3 www.csplib.org

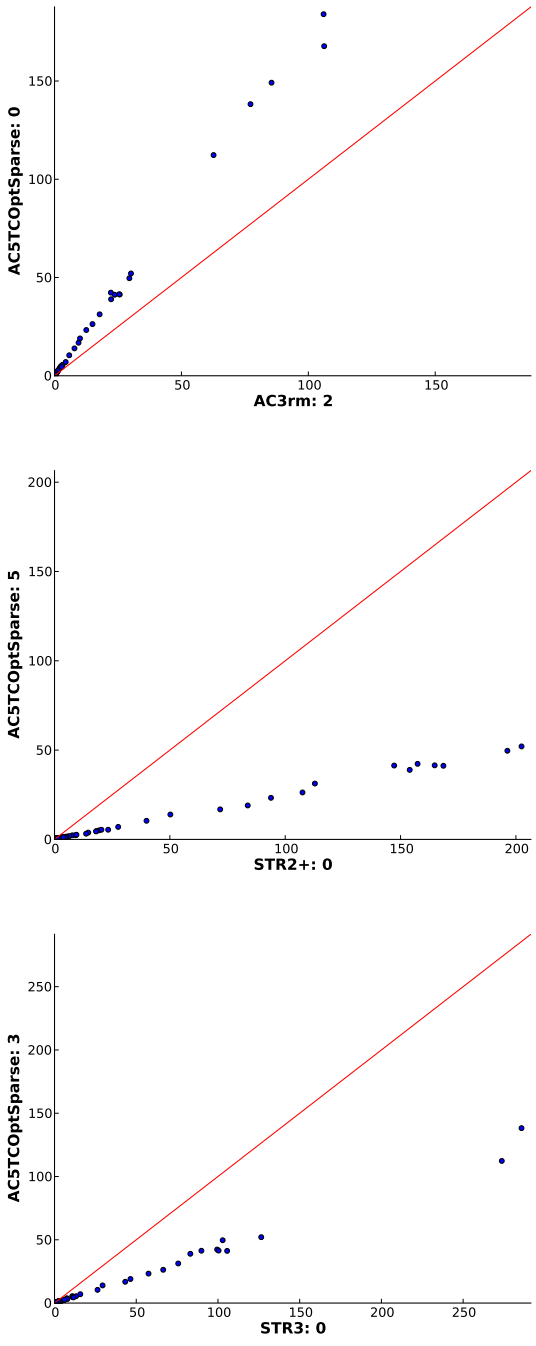


Figure 4.7: Scatter plots of the Geom instance set

propagator	totTime	postTime	nProp	%best	μ%best	valChk	QvalChk	pFollow
				$k = 2$				
GAC3-Allowed	16.3	0.6	1 M	315	324	166 k	0	166 k
AC5TC-Bool	18.6	0.8	2 M	358	342	576	178 k	316 k
AC5TC-Sparse	16.8	0.5	2 M	323	276	576	178 k	316 k
AC5TC-Recomp	10.1	0.4	2 M	195	199	27 k	0	154 k
AC5TCOpt-Tr	9.4	2.5	2 M	182	488	576	0	42 k
AC5TCOpt-Sparse	6.6	0.8	2 M	126	264	576	0	0
MDD^c	26.6	3.7	1 M	512	970	0	0	307 k
STR2+	26.7	1.3	1 M	514	643	46 k	0	0
STR3	23.7	1.6	2 M	456	638	576	0	0
AC3	11.3	0.2	1 M	218	176	0	0	0
AC3rm	**5.2**	0.1	1 M	100	101	0	0	0
				$k = 3$				
GAC3-Allowed	2.5	0.3	75 k	395	310	12 k	0	12 k
AC5TC-Bool	3.5	0.3	242 k	553	395	380	10 k	21 k
AC5TC-Sparse	2.5	0.2	242 k	398	324	380	10 k	21 k
AC5TC-Recomp	1.5	0.2	239 k	244	213	2 k	0	12 k
AC5TCOpt-Tr	2.2	0.9	242 k	342	402	380	0	4 k
AC5TCOpt-Sparse	1.4	0.4	242 k	227	242	380	0	0
MDD^c	3.9	1.5	75 k	608	718	0	0	22 k
STR2+	3.7	0.6	75 k	585	546	5 k	0	0
STR3	4.0	0.7	242 k	639	627	380	0	0
AC3	1.6	0.1	85 k	223	183	0	0	0
AC3rm	**0.7**	0.1	85 k	100	100	0	0	0
				$k = 4$				
GAC3-Allowed	23.4	1.3	419 k	477	379	19 k	0	19 k
AC5TC-Bool	42.5	1.6	1.6 M	867	524	677	20 k	36 k
AC5TC-Sparse	29.8	1.0	1.6 M	608	384	677	20 k	36 k
AC5TC-Recomp	17.0	0.8	1.58 M	347	244	3 k	0	18 k
AC5TCOpt-Tr	21.8	5.0	1.6 M	445	621	677	0	5 k
AC5TCOpt-Sparse	12.3	1.7	2 M	250	315	677	0	0
MDD^c	31.2	7.3	419 k	637	957	0	0	35 k
STR2+	33.2	3.3	419 k	677	676	10 k	0	0
STR3	39.3	3.4	2 M	802	730	677	0	0
AC3	11.7	0.4	419 k	238	188	0	0	0
AC3rm	**4.9**	0.2	419 k	100	103	0	0	0

Table 4.2: Experimental Results on Langford instances (times in seconds)

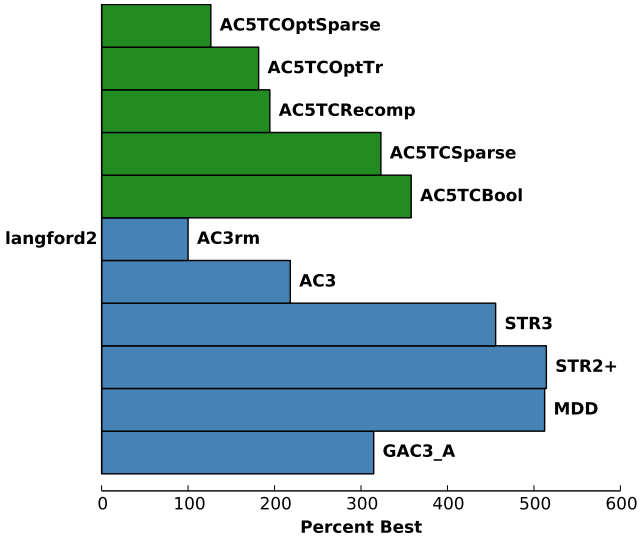


Figure 4.8: Percent best quantities for the langford 2 instance set

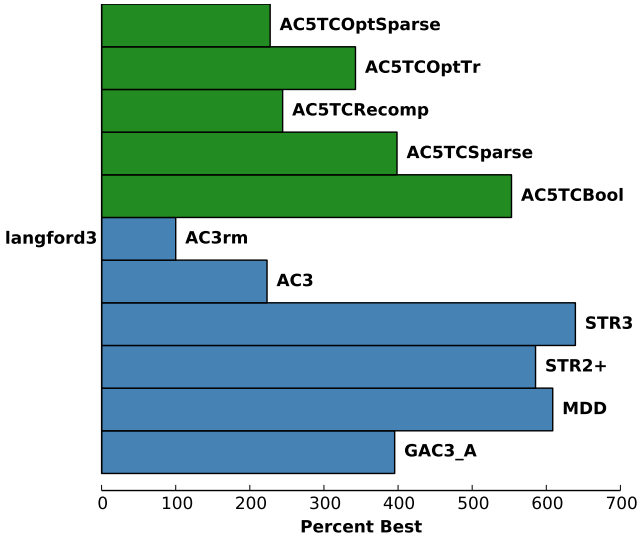


Figure 4.9: Percent best quantities for the langford 3 instance set

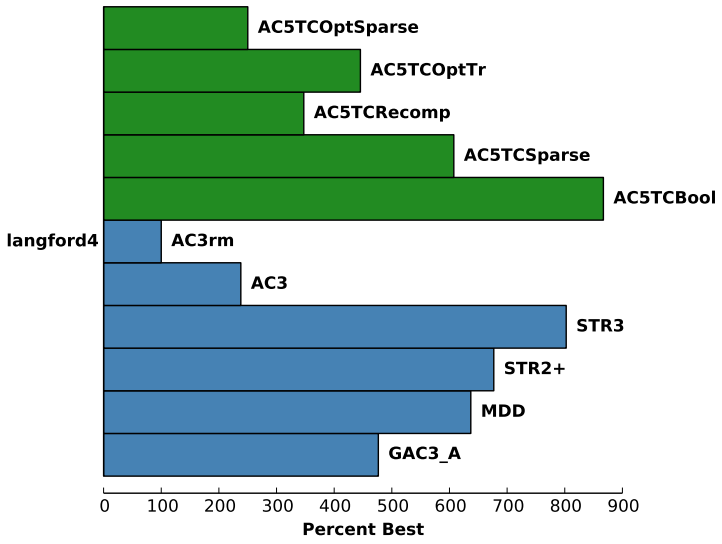


Figure 4.10: Percent best quantities for the langford 4 instance set

followed pointers is globally higher for this instance set, due to the inclusion of instances with larger n. The number of calls to the propagators during the search is also larger on the $k = 2$ set. This suggests that AC5TC-Tr requires harder instances (found in the $k = 2$ set) to amortize the cost of its data structures. AC5TCOpt-Sparse does not have this problem, thanks to its lesser need in backtrackable structures.

Scatter plots for the Langford problem are given for the $k = 4$ set in Figure 4.11. In those scatter plots are given, next to each algorithm, the number of instances that are solved by it and that caused the other algorithm to timeout. The scatter plots for the other values of k displayed the same patterns. The observations are similar to the ones made for the geom instance set: AC3rm seems linearly faster than AC5TCOpt-Sparse and AC5TCOpt-Sparse seems linearly faster than STR2+ and STR3. AC5TCOpt-Sparse and AC3rm solve one instance more than STR2+ and STR3.

The Traveling Salesman Problem We continue with the results of the propagators on the Traveling Salesman Problem (TSP) constraint satisfaction instances. We used the set of instances *tsp-20* and *tsp-25* [Lec]. Those structured instances are composed of very different table constraints. Their arity is 2 or 3. Some tables have up to 20,000 tuples, but some others have as few as 20. The variables also have quite different domain sizes: some have small

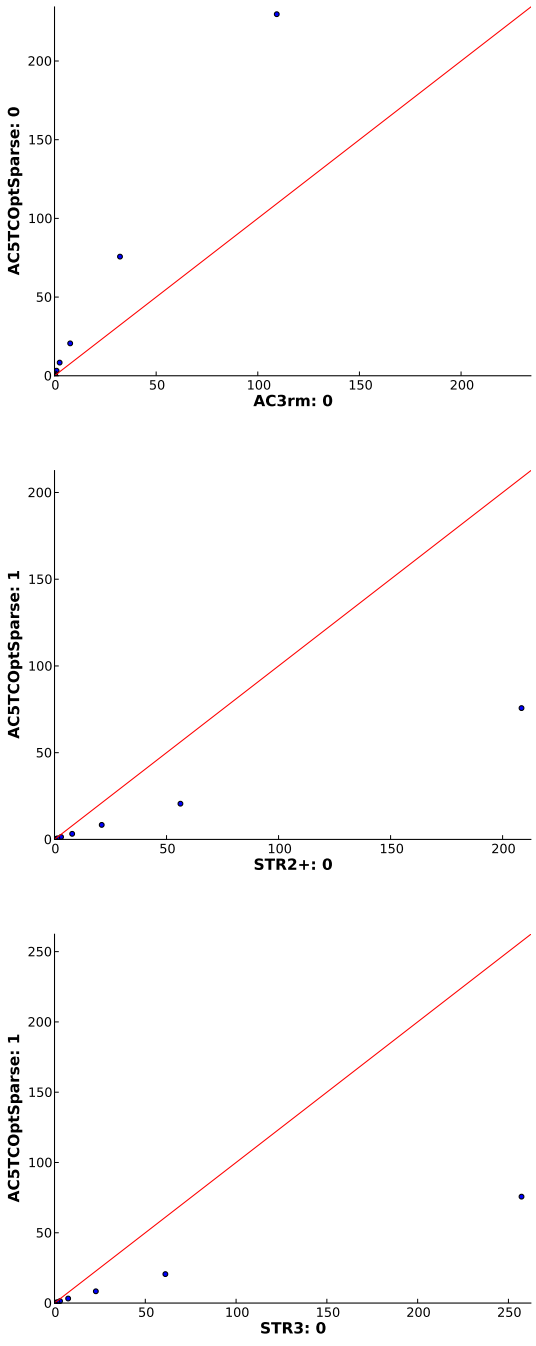


Figure 4.11: Scatter plots of the Langford instance set for $k = 4$

propagator	totTime	postTime	nProp	%best	μ%best	valChk	QvalChk	pFollow
GAC3-Allowed	797	1.7	6.7 M	1 073	795	11 M	0	11 M
AC5TC-Bool	186	0.8	21.2 M	251	254	2 k	1 M	2 M
AC5TC-Sparse	153	0.5	21.2 M	207	195	2 k	1 M	2 M
AC5TC-Recomp	109	0.3	20.9 M	146	140	391 k	0	1 M
AC5TCOpt-Tr	120	3.3	21.2 M	162	222	2 k	0	466 k
AC5TCOpt-Sparse	**74**	0.5	21 M	100	104	2 k	0	0
MDD^c	456	19.0	6.7 M	614	1041	0	0	7 M
STR2+	398	1.4	6.7 M	536	478	803 k	0	0
STR3	226	1.0	21 M	305	296	2k	0	0

Table 4.3: Results of the propagators for instance set TSP-20 (times in seconds)

domains, while others have domains containing up to 1000 values. There are 61 variables and 230 table constraints in *tsp-20* instances. The *tsp-25* instances have 76 variables and 350 constraints. The negative table constraints found in those instances have been transformed into positive ones by computation of their complement. The search strategy used here is *dom/deg* with lexicographic value ordering. Both sets contain 15 instances. For the set *tsp-25*, instance *tsp-25-715* has been removed from the set, as it was unsolved after three hours.

Tables 4.3 and 4.4 present the results. The plots of the percent best quantities are given in Figures 4.12 and 4.13. We first observe that STR2+, STR3 and MDD^c are slower than our propagators, except for the set TSP-25 where STR3 is faster than AC5TC-Bool. AC5TCOpt-Sparse is the fastest propagator on both instance sets. It is the best for each instance of the TSP-20 set. Another observation is that AC5TC-Recomp is faster than AC5TCOpt-Tr on the TSP-20 set, despite AC5TCOpt-Tr's being optimal. On the other hand, AC5TCOpt-Tr is faster on the TSP-25 set. We can also see that checking the validity instead of the Q-validity allows AC5TC-Recomp to follow fewer pointers and perform fewer validity checks than the Q-validity checks of AC5TC-Bool and AC5TC-Sparse. Moreover, on these instances, the small arity makes the validity check ($O(r)$) not so expensive compared to Q-validity ($O(1)$). This explains the good performance of AC5TC-Recomp.

To test the effect of the arity on this instance set, we merged binary tables of the instances of the TSP-20 set into arity 4 tables. The merge was obtained by merging arity 2 constraints into arity 4 ones. The merged constraints do

propagator	totTime	postTime	nProp	%best	μ%best	valChk	QvalChk	pFollow
GAC3-Allowed	6 607	2.4	73 M	931	764	23 M	0	23 M
AC5TC-Bool	2 625	1.3	198 M	370	350	2 k	11 M	19 M
AC5TC-Sparse	1 937	0.7	198 M	273	263	2 k	11 M	19 M
AC5TC-Recomp	1 315	0.5	196 M	185	180	3 M	0	10 M
AC5TCOpt-Tr	1 089	5.2	198 M	153	151	2 k	0	3 M
AC5TCOpt-Sparse	**710**	0.9	198 M	100	100	2 k	0	0
MDD^c	4 974	25.2	73 M	701	637	0	0	28 M
STR2+	3 740	2.9	73 M	527	500	5 M	0	0
STR3	2 308	1.9	198 M	325	305	2 k	0	0

Table 4.4: Results of the propagators for instance set TSP-25 (times in seconds)

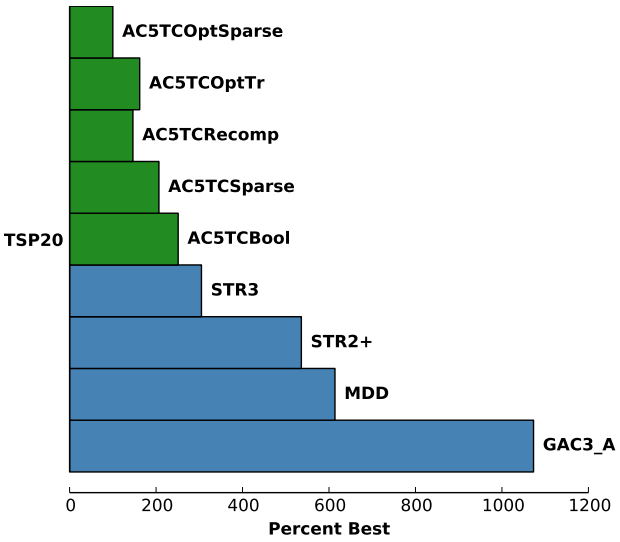


Figure 4.12: Percent best quantities for the TSP-20 instance set

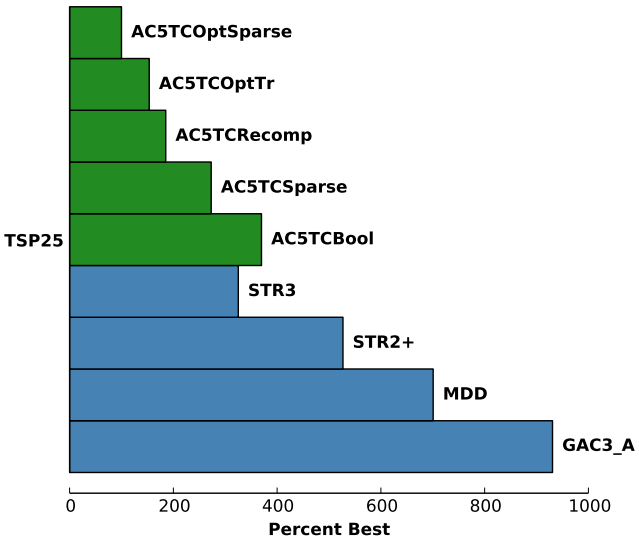


Figure 4.13: Percent best quantities for the TSP-25 instance set

not share variables. The selection of the constraints to merge was made by a greedy algorithm trying to maximize the number of merged constraints. The pruning in this benchmark is thus the same as in the original one. The results are summarized in Table 4.5. A plot of the percent best quantities is presented in Figure 4.14. A timeout of 15 minutes was set for these experiments. The data in the table concerns only instances for which none of the propagators timeouts. The percentage of the 15 instances solved by each propagator individually is also given. AC5TCOpt-Sparse is the fastest propagator, as for the TSP-20 and TSP-25 instance sets. However, STR2+ and STR3 are faster than our other propagators on those instances. This seems to indicate that those existing state-of-the-art propagators are better for larger arity constraints. The three propagators solving the largest number of instances are our two AC5TCOpt algorithms and STR2+.

Scatter plots for the TSP-20 and TSP-20 with quaternary tables are given in Figures 4.15 and 4.16. The patterns on the set TSP-25 are similar to the ones in the scatter plots of the TSP-20 set. For the modified TSP-20 set, next to each algorithm, is the number of instances solved by this algorithm for which the other timeouts. As we can see on those scatter plots, AC5TCOpt-Sparse is linearly better than STR2+ and STR3 on both instance sets. STR2+ is a bit faster on the instance set with quaternary tables while STR3 is disadvantaged.

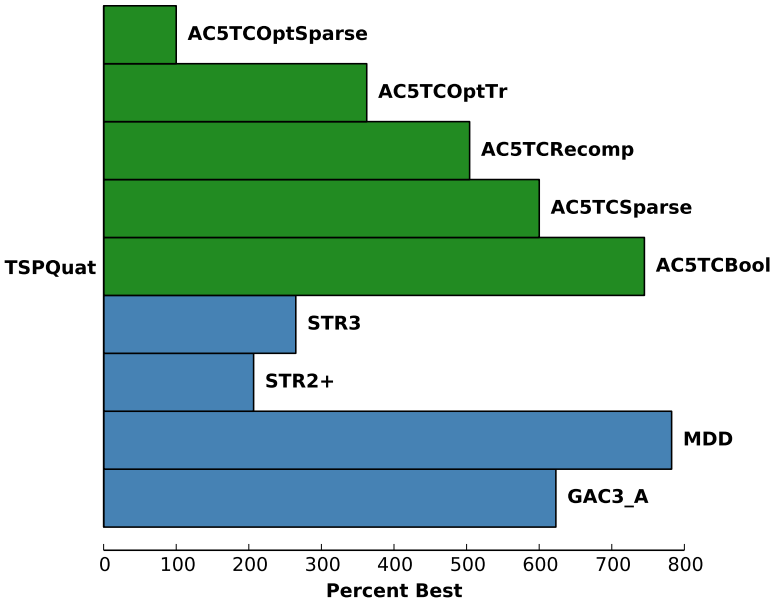


Figure 4.14: Percent best quantities for the quaternary TSP instance set

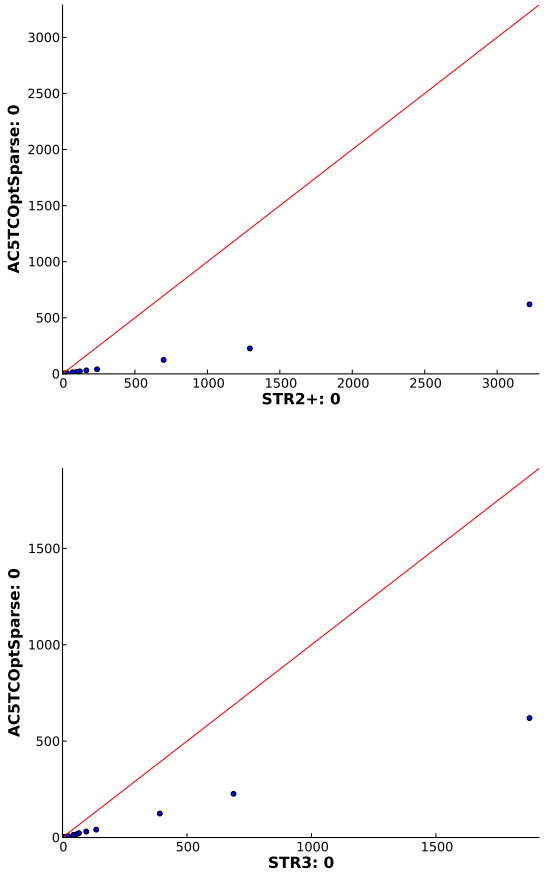


Figure 4.15: Scatter plots of the TSP-20 instance set

Two instances with quaternary tables are solved by AC5TCOpt-Sparse while triggering a timeout for STR3.

RandRegular The Regular constraint [Pes04] is a global constraint on a sequence of variables stating that the values taken by the variables have to form a word in a given regular language. The regular language is specified by a deterministic finite automaton. This constraint generalizes some other well known global constraints [Pes04]. Examples of problems where those constraints are heavily used in CP are rostering problems. In rostering, regular constraints are used to enforce the valid patterns of activities.

Each regular constraint can be encoded efficiently with a table constraint [BCDP05, QW06]. Since the length of the sequence is fixed at the number

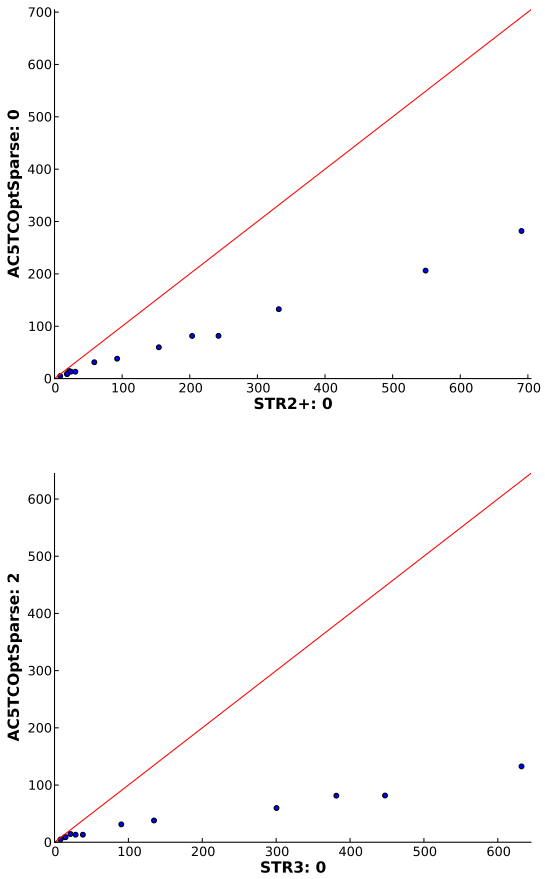


Figure 4.16: Scatter plots of the modified TSP-20 instance set with quaternary tables

propagator	totTime	postTime	nProp	%best	μ%best	%solved	valChk	QvalChk	pFollow
GAC3_Allowed	103	7.7	303 k	623	336	60	1 M	0	1 M
AC5TC-Bool	123	6.1	853 k	745	392	60	7 k	881 k	1 M
AC5TC-Sparse	99	4.0	853 k	600	303	67	7 k	881 k	1 M
AC5TC-Recomp	83	3.4	844 k	504	261	73	227 k	0	793 k
AC5TCOpt-Tr	60	44.1	853 k	362	429	93	7 k	0	36 k
AC5TCOpt-Sparse	**16.6**	7.3	852 k	100	102	93	7 k	0	0
MDD^c	130	104.0	303 k	782	934	87	0	0	456 k
STR2+	34.2	16.0	303 k	207	200	93	80 k	0	0
STR3	43.8	10.6	853 k	265	217	80	7 k	0	0

Table 4.5: Experimental Results on tsp-20 instances with arity 4 tables

of variables in the scope of the global constraint, additional variables can be introduced to represent the successive states visited in the automaton. For a regular constraint with scope $x_1 \dots x_r$, those state variables are $q_0 \dots q_r$. For all $0 \leq i < r$, a constraint links the variables in the scope and the additional variables $q_{i+1} = \mathit{Trans}(q_i, x_{i+1})$, where *Trans* is the transition function of the automaton. These constraints are posted using table constraints to encode the transition function. The tables are computed based on the transition function and the reachable states. Two additional constraints are posted: $q_0 = s$ and $q_n \in F$, where s is the starting state of the automaton and F is its set of final states.

For these experiments, we generated 100 instances with regular constraints. These instances contain 20 regular constraints on 10 different variables. Each regular constraint has a scope of 5 variables, chosen randomly. The domains of the variables are of size 10. Each regular constraint contains 20 states and has a randomly created transition table, hence the name of the benchmark: RandRegular. Amongst the states, 30% are randomly chosen to be final. The parameters were chosen to produce instances with a significant number of fails and choice points. The regular constraints were transformed into ternary table constraints. The search strategy used during the resolution was *dom/deg* variable ordering with lexicographic variable ordering.

The results of the experiments on the RandRegular instances can be found in Table 4.6. Figure 4.17 gives the percent best quantities for this instance set. On this instance set, all our propagators are faster than the existing state-of-the-

propagator	totTime	postTime	nProp	%best	μ%best	valChk	QvalChk	pFollow
GAC3-Allowed	68	0.1	2 M	205	200	1 M	0	1 M
AC5TC-Bool	40.5	0.0	8 M	122	121	137	462 k	955 k
AC5TC-Sparse	42.6	0.0	8 M	128	128	137	462 k	955 k
AC5TC-Recomp	36.4	0.0	8 M	110	109	288 k	0	766 k
AC5TCOpt-Tr	42.5	0.1	8 M	128	129	137	0	367 k
AC5TCOpt-Sparse	**33.2**	0.1	8 M	100	100	137	0	0
MDD^c	105	0.6	2 M	316	313	0	0	1 M
STR2+	96	0.0	2 M	288	283	371 k	0	0
STR3	69	0.1	8 M	208	207	137	0	0

Table 4.6: Experimental Results on RandRegular Instances

art ones. The winning strategy is our optimal AC5TCOpt-Sparse. Despite the large variability in resolution times for a single technique between different instances, the small differences between the *%best* and μ*%best* indicates that the performances of the propagators are proportional through the whole set. The arity of the tables (all tables have an arity of 3) as well as their medium size, which is constant through the set, could explain the good performance of our algorithms on this instance set.

Scatter plots for the RandReg benchmark are given in Figure 4.18. We can observe that the solving times are well spread for those instances. We can also see that AC5TCOpt-Sparse is linearly faster than both STR2+ and STR3, STR3 being faster than STR2+.

Random Instances These instances contain table constraints randomly generated by the RD-model [XBHL07]. The parameters were chosen to generate instances in or close to the phase transition, using Theorems 1 and 2 from [XBHL07]. The phase transition is the space where the generated instances transition from being trivially solvable to being trivially unsolvable. Instances in the phase transition are the hardest to solve. The instances have 10 variables, a uniform domain size of 10, and 15 table constraints of arity 5. The expected number of tuples in each table is 20,000. The 10 instances of the set were generated with those settings. The search strategy was the *dom* heuristic with lexicographic value ordering.

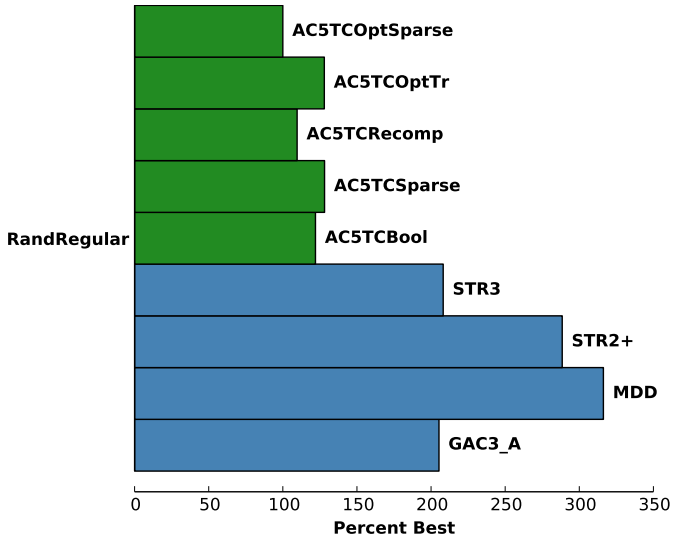


Figure 4.17: Percent best quantities for the RandRegular instance set

propagator	totTime	postTime	nProp	%best	μ%best	valChk	QvalChk	pFollow
GAC3-Allowed	3 000	1.5	614 k	2 725	2 660	523 M	0	523 M
AC5TC-Bool	4 636	1.0	2.8 M	4 211	4 070	19 k	257 M	481 M
AC5TC-Sparse	3 991	0.8	2.8 M	3 626	3 538	19 k	257 M	481 M
AC5TC-Recomp	3 874	0.8	2.4 M	3 519	3 357	98 M	0	305 M
AC5TCOpt-Tr	994	5.2	2.8 M	903	930	19 k	0	16 M
AC5TCOpt-Sparse	469	1.7	2.8 M	426	440	19 k	0	0
MDD^c	**110**	12.4	614 k	100	100	0	0	12 M
STR2+	483	0.7	614 k	439	455	22 M	0	0
STR3	913	2.1	2.8 M	829	839	19 k	0	0

Table 4.7: Results of the propagators on fully random instance set (times in seconds)

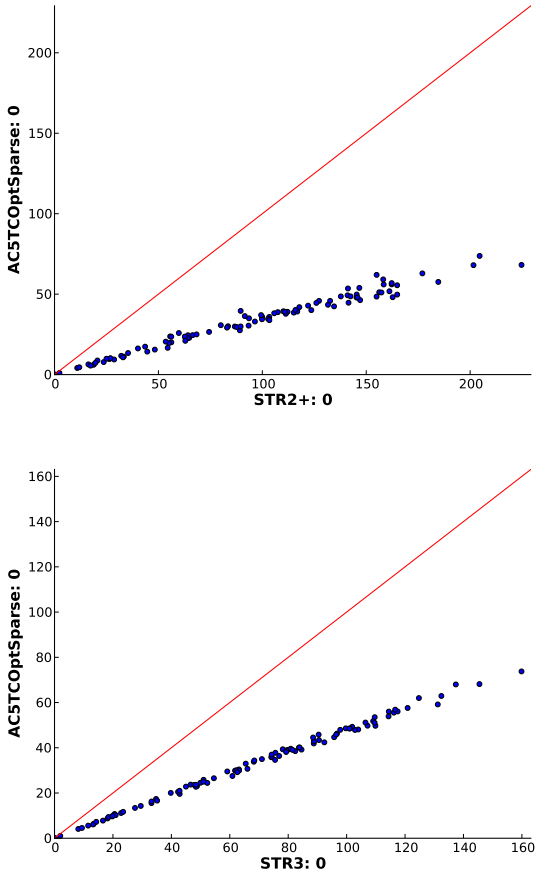


Figure 4.18: Scatter plots of the RandRegular instance set

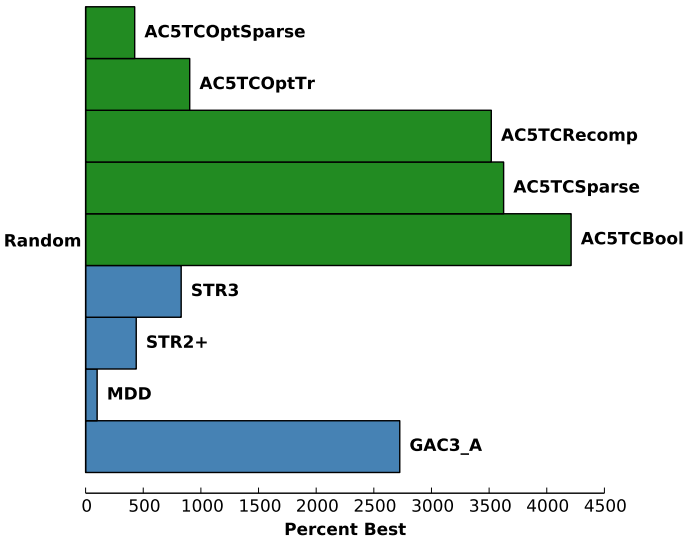


Figure 4.19: Percent best quantities for the Random instance set

Table 4.7 summarizes the results. Figure 4.19 plots the percent best quantities for this instance set. The results are similar for other parameter settings which also generate instances close to the phase transition. The standard MDD^c algorithm outperforms our value-based propagators on all instances, as it has a $\mu\%best$ of 100. The performance of our optimal AC5TCOpt-Tr is comparable to the performance of the optimal STR3 but AC5TCOpt-Sparse is significantly faster than both. AC5TCOpt-Sparse is the most efficient value based propagator. Observe the large number of validity checks of AC5TC-Recomp and Q-validity checks of AC5TC-Bool and AC5TC-Sparse, as well as the number of times they follow a pointer. The two AC5TCOpt implementations perform the same number of validity checks at post time as the two AC5TC ones, but they do not require any Q-validity checks afterwards. The difference in performance between AC5TCOpt-Tr and AC5TCOpt-Sparse reflects the additional cost AC5TCOpt-Tr has to pay for trailing its larger data structures. The performances of our propagators seems to suffer from the larger arity of the tables as well as their larger size. Indeed, AC5TC-Recomp, AC5TC-Sparse and AC5TC-Bool can revisit each tuple r times in the worst case and the trailable structures of AC5TCOpt-Tr are proportional to $r \cdot t$.

Figure 4.20 shows the scatter plots of AC5TCOpt-Sparse versus STR2+ and AC5TCOpt-Sparse versus STR3. In these graphs, we can see that STR2+ and

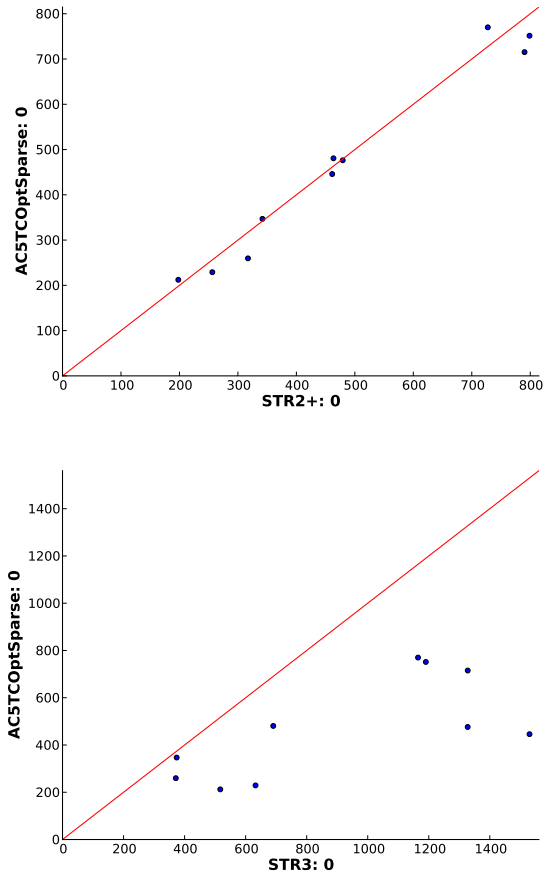


Figure 4.20: Scatter plots of the Random Benchmark

AC5TCOpt-Sparse perform similarly. On the other hand, AC5TCOpt-Sparse is faster than STR3.

The Crossword Problem The Crossword problem is the problem of filling a predefined grid with words from a dictionary. We used four instance sets from [Lec]. Those instances were also used to test table constraint propagators in [Lec11] and [LLY12]. The instances in those sets differ in which dictionary they use to get the words from. The grids are all the same, and empty. Inside a set, different grid sizes are used, varying the arity of the table constraints. The instance set *lexVg* uses the dictionary defined in [SS05]. Instances in *ogdVg* use a French dictionary. The set *ukVg* uses the UK cryptic solvers dictionary. The last instance set, *wordsVg*, uses the dictionary in /usr/dict/words under

Linux. The dictionaries used in *lexVg* and *wordsVg* are small, leading to small tables. In contrast, *ogdVg* and *ukVg* use large dictionaries, leading to large tables. Since, for these problems, the same word can be used several times, only table constraints are used to encode the problem.

The search heuristic used for this problem was *dom/deg* variable ordering with lexicographic value ordering. A timeout of 20 minutes was set on the resolution of the instances. The results concern only the instances for which none of the propagators timeouts. The grid sizes used in the experiments are:

- for *lexVg*, 42 instances: 4x{4..8}, 5x5, 6x6, 7x{10..11}, 8x{9..12}, 9x{9..13}, 10x{10..14}, 11x{11..15}, 12x{12..16}, 13x{13..17}, 14x{14..18}, 15x{15..18}, 16x{16..20}
- for *ogdVg*, 27 instances: 4x{4..8}, 5x{5..9}, 6x6, 7x7, 13x{16..17}, 14x{16..18}, 15x{15..19}, 16x{16..20}
- for *ukVg*, 23 instances: 4x{4..8}, 5x{5..7}, 6x{6}, 13x17, 14x{16..18}, 15x{15..19}, 16x{16..20}
- for *wordsVg*, 47 instances: 4x{4..8}, 5x{5..7}, 6x6, 7x11, 8x{11..12}, 9x{10..13}, 10x{10..14}, 11x{11..15}, 12x{12..16}, 13x{13..17}, 14x{14..18}, 15x{15..17}, 16x{16..18}

The four sets contain a total of 139 instances. The arity of each table is fixed by its grid size. An instance with a grid size of $x \times y$ has table constraints of arity x and y.

The results of the propagators on the Crossword instances can be found in Table 4.8. The percent best quantities are plotted in Figures 4.21, 4.22, 4.23, and 4.24, for, respectively, the sets of instances *lexVg*, *ogdVg*, *ukVg*, and *wordsVg*. Except for AC5TCOpt-Sparse on the *lexVg* set, both STR3 and STR2+ are faster than all the other propagators. However, the quantity $\mu\%best$ indicates that neither of them is the best on each instance. When compared to AC5TCOpt-Sparse, they are faster on the easiest and the hardest instances of the sets but AC5TCOpt-Sparse is faster on the instances of medium difficulty. On the sets *lexVg* and *ukVg*, the two AC5TC propagators have smaller $\mu\%best$ than STR2+ and STR3, meaning that they are generally closer to being the best, instance by instance. Another observation is that although AC5TCOpt-Tr is optimal, the non-optimal AC5TC-Bool and AC5TC-Sparse are faster on the four sets of crossword instances. On the *lexVg* and the *wordsVg* sets, AC5TCOpt-Sparse is the best of our propagators. Surprisingly, on the *ogdVg* and *ukVg*, the best of our propagators is AC5TC-Sparse, followed by AC5TC-Bool. Those

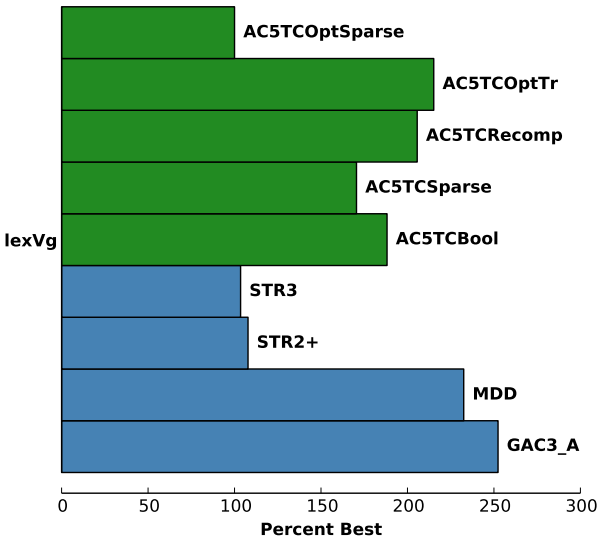


Figure 4.21: Percent best quantities for the *lexVg* instance set

two instance sets are the ones where the dictionaries used are the largest, resulting in larger tables. The cost of its large maintained structures seems to disadvantage AC5TCOpt-Tr on this problem. This characteristic of this benchmark, with large arity tables, seems to disadvantage our propagators.

Scatter Plots for *ogdVg* and *wordsVg* are given in Figures 4.25 and 4.26, respectively. The integer next to each algorithm is the number of instances solved by it triggering a timeout for the other algorithm. These two instance sets have been chosen because *ogdVg* instances have a large dictionary and *wordsVg* instances have a small one. For *ogdVg*, STR2+ is faster than AC5TCOpt-Sparse except on some instances where their performance is comparable. The STR3 propagator is clearly faster than AC5TCOpt-Sparse except on three non-easy instances, where AC5TCOpt-Sparse is significantly faster than STR3. AC5TCOpt-Sparse is able to solve one instance that STR3 can't within the time limit. On *wordsVg*, the performances of STR2+ and AC5TCOpt-Sparse are comparable, STR2+ being faster in average. STR3 is also faster than AC5TCOpt-Sparse on this instance set (except for two non-easy instances) but the relative difference is smaller here than in *ogdVg*. AC5TCOpt-Sparse is able to solve one *wordsVg* instance that caused both STR2+ and STR3 to timeout.

propagator	totTime	postTime	nProp	%best	μ%best	valChk	QvalChk	pFollow
				lexVg				
GAC3-Allowed	61	0.2	35 k	234	283	9 M	0	9 M
AC5TC-Bool	44.9	0.1	611 k	171	150	2 k	5 M	7 M
AC5TC-Sparse	41.2	0.1	611 k	157	139	2 k	5 M	7 M
AC5TC-Recomp	50	0.1	543 k	192	170	4 M	0	6 M
AC5TCOpt-Tr	66	0.6	611 k	252	378	2 k	0	847 k
AC5TCOpt-Sparse	30.3	0.2	611 k	116	191	2 k	0	0
MDD^c	77	7.8	35 k	293	1321	0	0	3 M
STR2+	31.8	0.2	35 k	121	160	957 k	0	0
STR3	**26.2**	0.2	611 k	100	167	2 k	0	0
				ogdVg				
GAC3-Allowed	55	2.9	4 k	264	263	6 M	0	6 M
AC5TC-Bool	41.5	1.2	61 k	200	168	12 k	3 M	4 M
AC5TC-Sparse	37.7	1.0	61 k	182	151	12 k	3 M	4 M
AC5TCRecomp	53	0.9	51 k	254	182	2 M	0	4 M
AC5TCOpt-Tr	122	14.7	61 k	589	563	12 k	0	348 k
AC5TCOpt-Sparse	52	2.8	61 k	249	212	12 k	0	0
MDD^c	146	82.0	4 k	704	1703	0	0	1 M
STR2+	33.6	1.6	4 k	162	180	548 k	0	0
STR3	**20.7**	3.1	61 k	100	151	12 k	0	0
				ukVg				
GAC3-Allowed	76	1.3	14 k	244	290	6 M	0	6 M
AC5TC-Bool	67	0.6	373 k	216	243	7 k	4 M	6 M
AC5TC-Sparse	58	0.5	373 k	186	212	7 k	4 M	6 M
AC5TC-Recomp	83	0.4	292 k	267	275	3 M	0	5 M
AC5TCOpt-Tr	175	4.3	373 k	565	482	7 k	0	506 k
AC5TCOpt-Sparse	77	1.2	372 k	247	203	7 k	0	0
MDD^c	222	56.0	14 k	713	1110	0	0	3 M
STR2+	41.9	0.6	14 k	135	128	583 k	0	0
STR3	**31.1**	1.3	373 k	100	150	7 k	0	0
				wordsVg				
GAC3-Allowed	63	0.3	20 k	246	296	8 M	0	8 M
AC5TC-Bool	49.5	0.2	362 k	193	161	2 k	5 M	6 M
AC5TC-Sparse	43.9	0.1	362 k	171	148	2 k	5 M	6 M
AC5TC-Recomp	56	0.1	317 k	219	173	3 M	0	5 M
AC5TCOpt-Tr	88	1.1	362 k	345	466	2 k	0	734 k
AC5TCOpt-Sparse	39.6	0.3	362 k	155	207	2 k	0	0
MDD^c	84	11.8	20 k	328	1840	0	0	2 M
STR2+	35.4	0.3	20 k	138	180	921 k	0	0
STR3	**25.6**	0.4	362 k	100	169	2 k	0	0

Table 4.8: Experimental Results on Crosswords instances

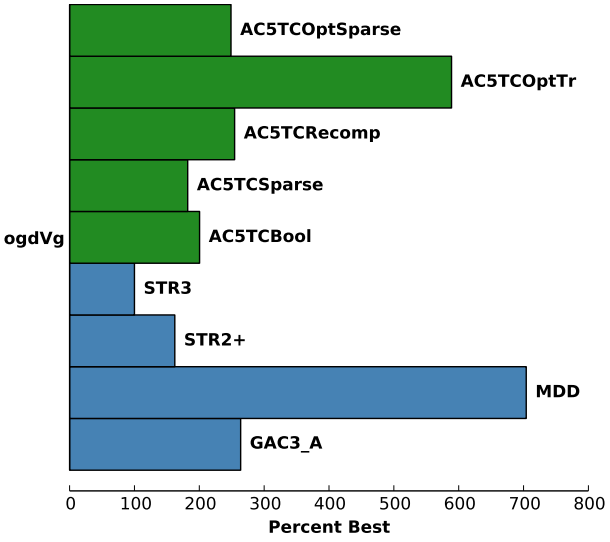


Figure 4.22: Percent best quantities for the *ogdVg* instance set

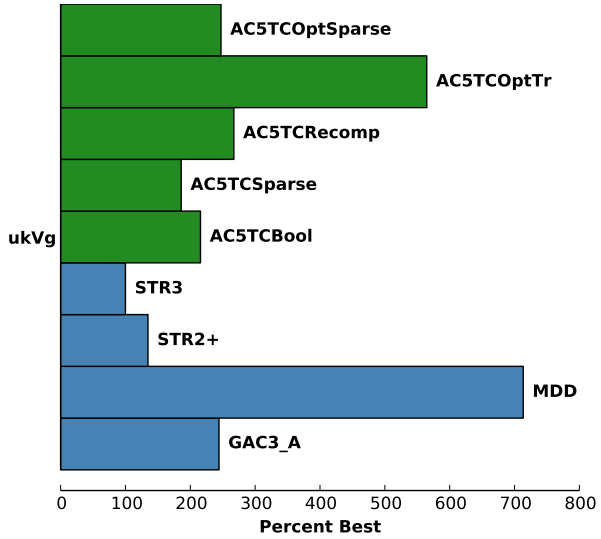


Figure 4.23: Percent best quantities for the *ukVg* instance set

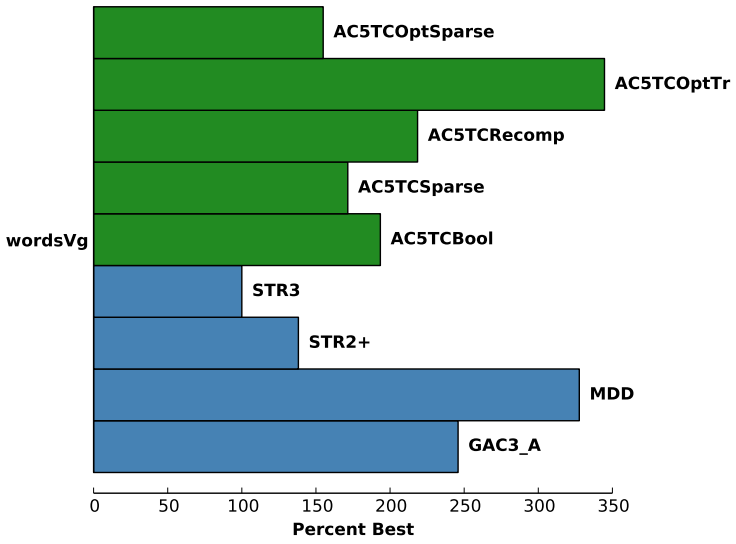


Figure 4.24: Percent best quantities for the *wordsVg* instance set

The Modified Renault Problem The modified Renault problem instances originate from a Renault Megane configuration problem. This problem has been modified in order to generate a series of instances. These instances have large tables (up to 50 k tuples) of large arities (up to arity 10). These instances can be found in [Lec]. The search strategy used was *dom/deg* variable ordering with lexicographic value ordering. A timeout of 20 minutes was set. The results concern only the instances for which none of the propagators timed out. The set of instances found on [Lec] has 50 instances. Amongst them, 16 were solved by all propagators within the time limit. The percentage of those 50 instances solved by each individual propagator is also given in the table.

The experimental results for the Modified Renault Problem are given in Table 4.9. Figure 4.27 plots the percent best quantities for this instance set. STR2+ is the fastest propagator on this instance set and it is the fastest on each instance. Our optimal AC5TCOpt-Sparse is faster than STR3 and it solves one instance more. However, STR3 is faster than our other propagators. Observe the difference in the number of calls to the propagator between the value based and constraint based propagators, giving advantage to the constraint based approaches. The difference in the number of calls between AC5TCOpt-Sparse and the group AC5TCOpt-Tr, AC5TC-Sparse and AC5TC-Bool comes from the order in which the tuples are visited during propagation. Indeed, all our

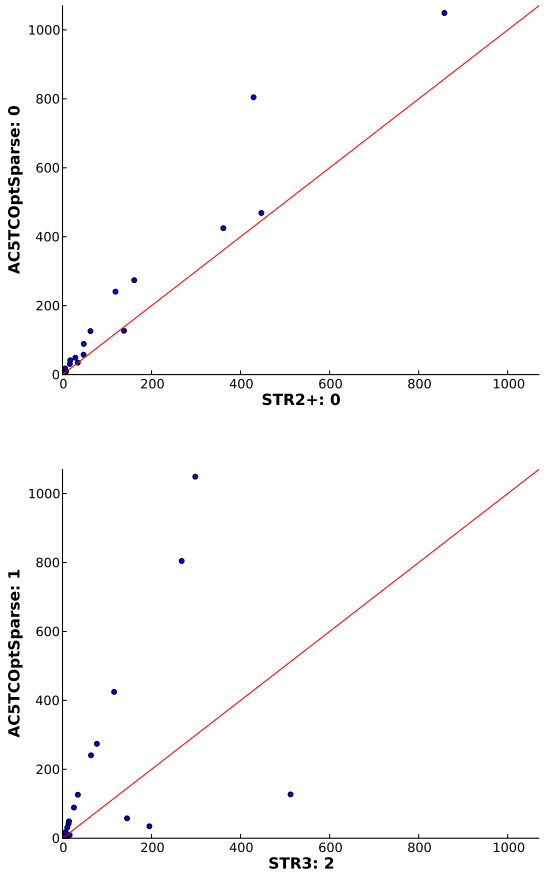


Figure 4.25: Scatter plots of the crossword instance set *ogdVg*

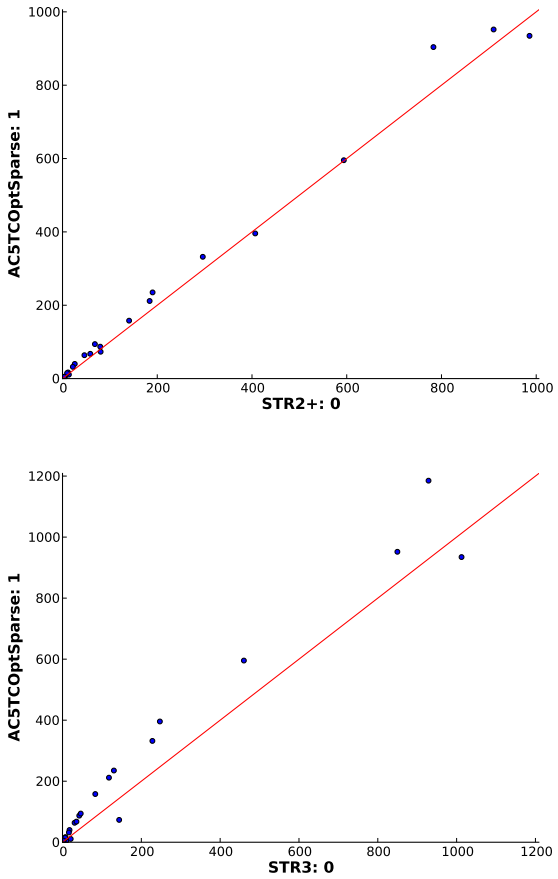


Figure 4.26: Scatter plots of the crossword instance set *wordsVg*

propagator	totTime	postTime	nProp	%best	μ%best	%solved	valChk	QvalChk	pFollow
GAC3-Allowed	16.5	1.4	39 k	307	215	34	286 k	0	286 k
AC5TC-Bool	14.4	0.8	268 k	268	202	32	1 k	128 k	221 k
AC5TC-Sparse	13.2	0.6	268 k	244	178	32	1 k	128 k	221 k
AC5TC-Recomp	12.1	0.5	227 k	224	174	32	67 k	0	155 k
AC5TCOpt-Tr	14.5	4.5	268 k	269	389	36	1 k	0	33 k
AC5TCOpt-Sparse	7.6	1.6	275 k	141	176	36	1 k	0	0
MDD^c	17.9	15.4	39 k	332	642	36	0	0	21 k
STR2+	**5.4**	0.6	39 k	100	100	36	25 k	0	0
STR3	10.4	1.7	268 k	193	198	34	1 k	0	0

Table 4.9: Experimental Results on the modified Renault problem

propagators visit the tuples in the order of the table, except AC5TCOpt-Sparse. This results in a difference in the order of the values in the propagation queue, and hence a difference in the number of calls to the propagators. AC5TC-Recomp has less calls due to its use of the validity, allowing it to compute a larger $\triangle$. Although the large arity of the tables seems to slow our propagators, AC5TCOpt-Sparse is the less impacted.

Scatter plots for these instances can be found in Figure 4.28. Next to each algorithm is the number of instances solved by it that the other algorithm was unable to solve within the time limit. Although the spreading of the solving time is not good, these plots confirm the tendency exhibited by the average solving time in the table: STR2+ is faster than AC5TCOpt-Sparse, and AC5TCOpt-Sparse is faster than STR3. We can also observe that AC5TCOpt-Sparse solves one instance that STR3 was unable to solve within the time limit.

Summary Table 4.10 gives a summary of the per benchmark percent to the best mean time. The names of our propagators have been shortened by removing the prefix 'AC5TC'. GAC3_Allowed has been shortened to 'GAC3_A'. This table shows the effect of the arity of the table constraints on the propagators. The benchmarks are put into three categories, depending on the arity of the table constraints: binary benchmarks, small arity benchmarks (arities 3 and 4), and large arity benchmarks. For the benchmarks containing only binary table constraints, AC3rm is clearly the fastest propagator. However, on those benchmarks, our propagators are generally faster than the existing state-of-the-

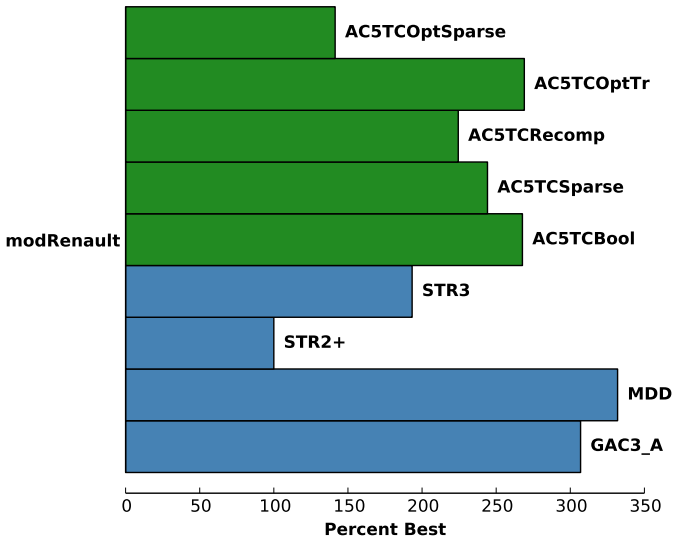


Figure 4.27: Percent best quantities for the modified Renault instance set

art MDD^c, STR2+ and STR3. AC3rm has been designed for binary constraints. For the benchmarks where the tables have arities of up to 4, our propagators are globally the fastest propagators. However, when the arity of the tables in the benchmarks increases, our propagators become slower than the state of the art. The existing state-of-the-art propagators considered in this chapter are well suited for problems where the arity is large. The MDD^c propagator is the fastest on the random instances. On this set, despite the random tables, its compressed table is small, allowing it to outperform the other propagators. The optimal STR3 propagator is the fastest on the crossword instances. STR2+ is the fastest on the modified Renault benchmark. Although it would be statistically meaningless to average the %*best* in Table 4.10, it is clear that on these benchmarks, our optimal AC5TCOpt-Sparse is the fastest among our propagators. It stays competitive with AC3rm on half of the binary benchmarks, it is globally the best on the small arity benchmarks, and on some of the large arity benchmarks, its performances are competitive with the state of the art. The non-optimal AC5TC-Recomp and the optimal propagator AC5TCOpt-Tr are the next fastest ones of our propagators. AC5TC-Tr outperforms AC5TC-Recomp on difficult instances. However, on easier instances, the cost of its trailable *nextTr* data structure makes it slower than AC5TC-Recomp. AC5TC-Bool and AC5TC-Sparse are generally slower than AC5TC-Recomp since they

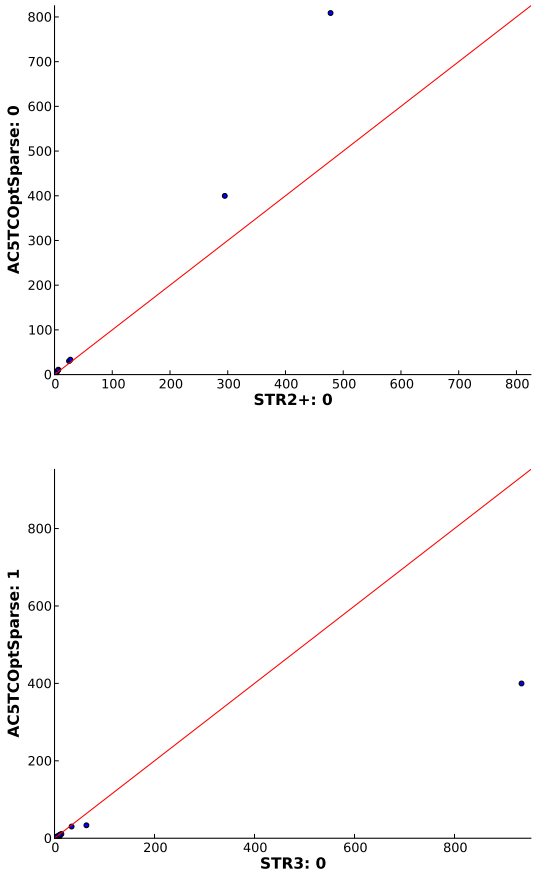


Figure 4.28: Scatter plots of the modified Renault instance set

Benchmark	GAC3_A	Bool	Sparse	Recomp	OptTr	OptSparse	MDD^c	STR2+	STR3	AC3	AC3rm
geom	276	341	295	216	263	178	401	680	413	283	**100**
Langford(2)	315	358	323	195	182	126	512	514	456	218	**100**
Langford(3)	395	553	398	244	342	227	608	585	639	223	**100**
Langford(4)	477	867	608	347	445	250	637	677	802	238	**100**
TSP-20	1 073	251	207	146	162	**100**	614	536	305	-	-
TSP-25	931	370	273	185	153	**100**	701	527	325	-	-
TSP-Quat-20	623	745	600	504	362	**100**	782	207	265	-	-
RandRegular	205	122	128	110	128	**100**	316	288	208	-	-
Random	2 725	4 211	3 626	3 519	903	426	**100**	439	829	-	-
CW-LexVg	234	171	157	192	252	116	293	121	**100**	-	-
CW-ogdVg	264	200	182	254	589	249	704	162	**100**	-	-
CW-ukVg	244	216	186	267	565	247	713	135	**100**	-	-
CW-wordsVg	246	193	171	219	345	155	328	138	**100**	-	-
modified Renault	307	268	244	224	269	141	332	**100**	193	-	-

Table 4.10: Summary of the experimental results: %best

test Q-validity, not validity, and hence make smaller jumps in the table. However, on the crossword instances, the two AC5TC algorithms are faster than AC5TC-Recomp on all instance sets and even than AC5TCOpt-Sparse on two of the four sets. Also, AC5TCOpt-Sparse is always faster than AC5TCOpt-Tr, confirming the additional cost AC5TCOpt-Tr has to pay for its backtrackable structures. AC5TC-Sparse is also always faster than AC5TC-Bool, for the same reasons.

4.6 STATISTICAL TREATMENT OF THE EXPERIMENTS

This section is concerned with the application of the statistical procedure presented in Chapter 3 to the experimental results for the propagators in this chapter. In this context, as timeouts are present and there are several classes of instances, we will use the θ_5 statistic of interest. Recall that θ_5 is the mean width of the area between the multi-class cumulative distributions of the algorithms being compared. This represents the mean, over the different proportions of the whole instance set, of the differences of the times the algorithms take to solve the given proportions of the set. In multi-class empirical distributions, each

class of instances has a weight. The weights used in this section are designed to give similar importances to each type of set of instances, with the exception of the non-academic data sets (modified Renault and TSP), which receive more importance, and the fully random data sets, which receive less importance. For instance, each Langford dataset (Langford 2,3 and 4) gets one-third of the weight given to the geom dataset, in order to give the same importance to the Langford type as to the geom one. Larger datasets do not have a larger impact than small ones, as the number of instances in each dataset appears as the denominator of the contribution of the dataset in $\hat{\theta}_5$. In order to illustrate the bootstrapped results, the non-bootstrapped multi-class empirical cumulative distributions of the algorithms are given in Figures 4.29 and 4.30. Recall that the multi-class empirical cumulative distribution for an algorithm A gives, for each time t, the weighted proportion of the full instance set solved by A in a time less than or equal to t. Figure 4.29 gives the MECDs of all the algorithms developed in this chapter. In this figure, we can see that the two optimal propagators, AC5TCOpt-Tr and AC5TCOpt-Sparse, are clearly faster than the other non-optimal ones. AC5TCOpt-Sparse is globally the fastest and it is the one that solves the largest proportion of instances at the end. Among the non-optimal propagators, AC5TC-Recomp seems to be the fastest. This may be explained by the fact that the structures that AC5TC-Recomp is able to reach the (same) fixed point faster by performing more pruning at each call of the propagator. Figure 4.30 gives the MECDs of our best propagator (AC5TCOpt-Sparse) versus the state of the art. As we can see, AC5TCOpt-Sparse is globally the fastest propagator and solves the largest proportion of instances at the end. However, for small solving times, the MDD^c propagator seems to be faster than AC5TCOpt-Sparse. Among the state of the art, MDD^c is the fastest for small solving times and STR3 becomes the faster for large solving times.

Bootstrapping was used on the estimator of θ_5 to get a confidence interval. The results of the pairwise comparisons of the algorithms are given in Table 4.31. The confidence intervals are written with the lower bound on top of the upper bound. To obtain these confidence intervals, 10,000 bootstrap samples were drawn from the full set of experiments and a confidence level α of 0.05 was used. The numbers correspond to seconds. The entry in the table in the line of algorithm A and the column of algorithm B gives the confidence interval on $\theta_5(A, B)$. Positive values correspond to A's being faster than B in general (all the instances included). If 0 is outside the confidence interval for $\theta_5(A, B)$, then A and B have significant performance differences. As AC3 and AC3rm are designed for binary constraints only, they are left out of the comparison. The convention for naming the algorithms is the same as the one used in Table

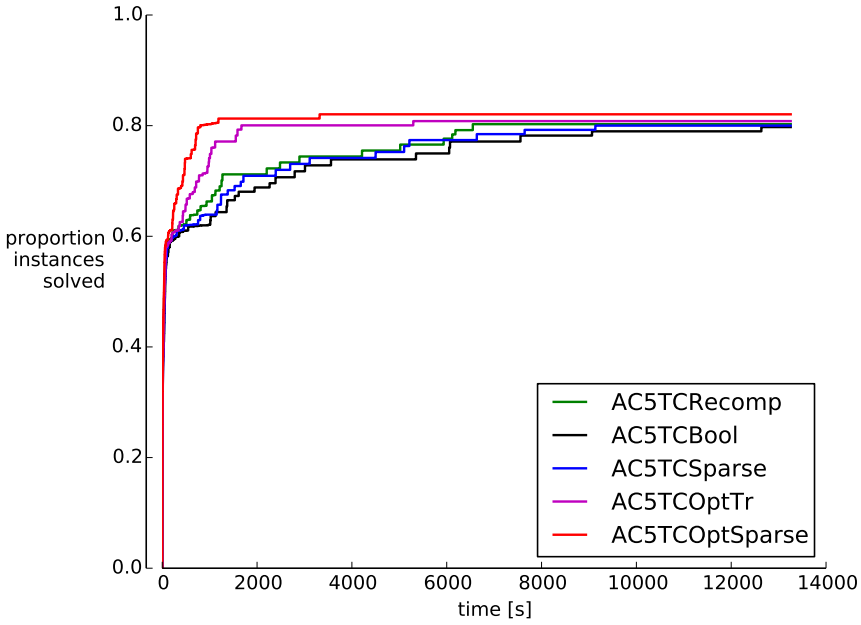


Figure 4.29: Non bootstrapped multi-class empirical cumulative distributions of the algorithms developed in this chapter

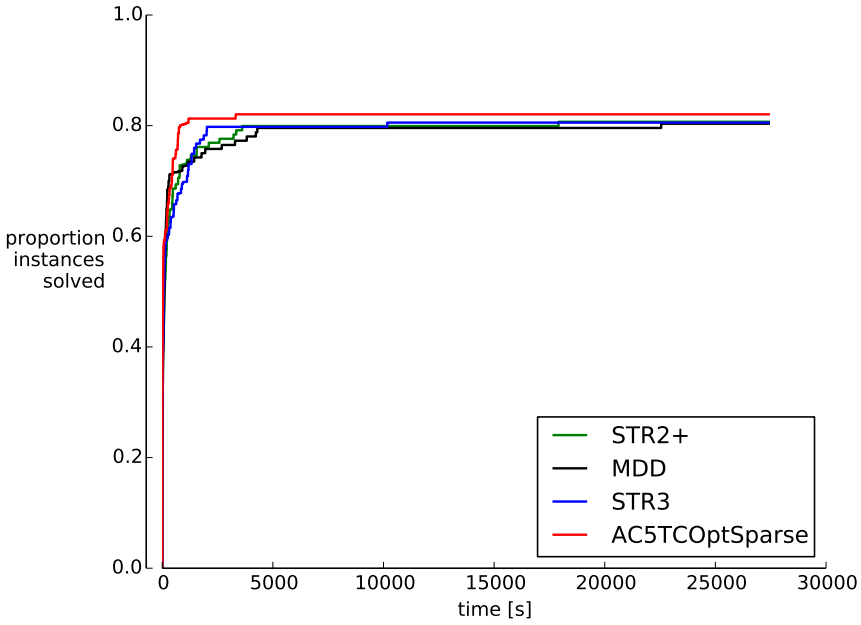


Figure 4.30: Non bootstrapped multi-class empirical cumulative distributions of AC5TCOptSparse and the state of the art

	GAC3_A	Recomp	Bool	Sparse	OptTr	STR2+	MDD	STR3	OptSparse
GAC3_A	–	-936.5	-339.6	-684.5	-1476.5	-1246.4	-1025.5	-1304.5	-1624.3
		29	441.6	163.1	-349.7	-337.6	-360.3	-320	-404.9
Recomp	-29	–	**124.9**	**12.2**	-725.7	-639.7	-530.5	-583.2	-874.5
	936.5		**462.6**	**220.2**	-242.3	-24	214.9	-124.4	-329.7
Bool	-441.6	-462.6	–	-294.9	-1113.6	-1015.1	-987.8	-1018.8	-1244.7
	339.6	-124.9		-98.4	-425.3	-383.4	-292.1	-417.9	-476.3
Sparse	-163.1	-220.2	**98.4**	–	-873.5	-776.5	-715.9	-735.6	-994
	684.5	-12.2	**294.9**		-321.3	-224.7	-31.8	-264	-404.6
OptTr	**349.7**	**242.3**	**425.3**	**321.3**	–	-14.5	-83	**11.9**	-220.7
	1476.5	**725.7**	**1113.6**	**873.5**		440.7	608.9	**236.3**	-90.8
STR2+	**337.6**	**24**	**383.4**	**224.7**	-440.7	–	-153.9	-200.8	-632.2
	1246.4	**639.7**	**1015.1**	**776.5**	14.5		305.6	108.5	-62.9
MDD	**360.3**	-214.9	**292.1**	**31.8**	-608.9	-305.6	–	-426.9	-764.4
	1025.5	530.5	**987.8**	**715.9**	83	153.9		137.8	-23.5
STR3	**320**	**124.4**	**417.9**	**264**	-236.3	-108.5	-137.8	–	-408.4
	1304.5	**583.2**	**1018.8**	**735.6**	-11.9	200.8	426.9		-101.7
OptSparse	**404.9**	**329.7**	**476.3**	**404.6**	**90.8**	**62.9**	**23.5**	**101.7**	–
	1624.3	**874.5**	**1244.7**	**994**	**220.7**	**632.2**	**764.4**	**408.4**	

Figure 4.31: Confidence intervals for the θ_5 statistic of interest on the GAC propagators for table constraints. The values correspond to seconds and the confidence intervals are given with the lower bound on top of the upper bound.

4.10. In bold are the confidence intervals that are significant (0 outside the interval) and positive (algorithm in the row is faster).

In Table 4.31, we can see that AC5TCOpt-Sparse is significantly faster than all the other propagators. Our propagators AC5TCOpt-Tr and AC5TCOpt-Sparse (optimal time complexity) are significantly faster than all our other propagators (non-optimal time complexities) and than the STR3 propagator. The only optimal propagator significantly faster than MDD^c and STR2+ is our AC5TCOpt-Sparse. The confidence intervals for the other optimal propagators versus MDD^c and STR2+ contain 0. GAC_allowed is significantly slower than all the algorithms except our non-optimal propagators (AC5TC-Recomp, AC5TC-Bool and AC5TC-Sparse), as 0 is in the confidence interval. Despite recomputing information, AC5TC-Recomp is significantly faster than all our propagators that do not have an optimal time complexity. The non-optimal STR2+ is significantly faster than all our non-optimal propagators.

The non-optimal MDD^c propagator is significantly faster than AC5TC-Bool and AC5TC-Sparse, but not AC5TC-Recomp. The optimal STR3 propagator is only significantly slower than our optimal propagators AC5TCOpt-Tr and AC5TCOpt-Sparse.

5

THE SMART TABLE CONSTRAINT

The biggest problem with table constraints are their size and lack of structure. Several approaches have been proposed to cope with their sizes (see Section 2.2). Among those approaches, some propose an alternative representation of the tables that involves compression. Such propagators are called compression-based. Some compression-based propagators propose modifying the definition of the tuples inside the table to obtain tables with fewer tuples. Compressed tuples [KW07, Rég11, XY13] introduce sets of values inside the tuples. A single compressed tuple represents all the tuples in the Cartesian product of the sets. This modification of the definition of tuples reduces the sizes of the tables and re-introduces structure inside the table constraints: similar tuples can be compressed into one compressed tuple. Another compression-based propagator [JN13], applying short supports to table constraints, allows variables to be left out of the tuples. A single short support tuple represents all the tuples where left-out variables take any values of their domains, the other variables taking the same values. Again, the tables are shorter and structure is re-introduced into the tables. These two modifications of the definition of tuples allow the development of efficient propagators, using their definition of tuples to advantage. In this chapter, we propose to generalize both compressed tuples and short supports in table constraints by introducing simple

arithmetic constraints inside the tuples. We call such tuples *smart tuples*, and tables containing smart tuples *smart tables*. For instance, the following set of tuples $\{(1,2,1), (1,3,1), (2,2,2), (2,3,2), (3,2,3), (3,3,3)\}$ on the variables $\{x_1, x_2, x_3\}$ with domains $\{1,2,3\}$ can be represented by a smart table containing only one smart tuple:

x_1	x_2	x_3
$= x_3$	≥ 2	$*$

or in an equivalent form by $(x_1 = x_3, x_2 \geq 2)$. A symbol $*$ in the tabular form of a smart tuple means that if not occurring anywhere else, the corresponding variable is not constrained at all by the tuple (which is not the case here).

To motivate the employment of smart tables, let us consider a car configuration problem. We assume that the cars to be configured have two colors (one for the body, *colB*, and the other for the roof, *colR*), a model number *modNum*, an option pack *optPack*, and an onboard computer *comp*. A configuration rule might state that, for a particular model number a and some fancy body color set S, an option pack less than a certain pack b implies that the onboard computer cannot be the most powerful one, c, and that the roof color has to be the same as the body color. This configuration constraint can be written as

$$modNum = a \wedge colB \in S \wedge optPack < b$$
$$\Rightarrow$$
$$comp \neq c \wedge colR = colB$$

The encoding of this constraint with a smart table consists of four smart tuples: $(modNum \neq a), (colB \notin S), (optPack \geq b)$ and $(comp \neq c, colR = colB)$, which gives under tabular form:

modNum	*colB*	*colR*	*optPack*	*comp*
$\neq a$	$*$	$*$	$*$	$*$
$*$	$\notin S$	$*$	$*$	$*$
$*$	$*$	$*$	$\geq b$	$*$
$*$	$*$	$= colB$	$*$	$\neq c$

Encoding this constraint with classical tuples is exponentially larger, and even using compressed tuples or short supports results in a table that is strictly

longer because none of these techniques can be used to encode compactly the relation existing between *colB* and *colR* (they require, for this case, one distinct tuple for each possible color). On the other hand, using reification (decomposition by adding auxiliary variables) of the configuration rule does not guarantee the same level of pruning as the smart table encoding, since there is a cycle to handle.

Related Publication

> [MDL15] Jean-Baptiste Mairy, Yves Deville and Christophe Lecoutre, "The Smart Table Constraint", Integration of AI and OR Techniques in Constraint Programming (CPAIOR) 2015

5.1 SYNTAX AND SEMANTICS

A *table constraint* is a constraint whose semantics is defined in extension by listing the set of allowed (or forbidden) tuples. These tuples are *classical*. In this chapter, we introduce smart table constraints. A *smart table constraint* sc is defined semantically from a set of smart tuples, called a *smart table* and denoted by $table(sc)$. A *smart tuple* σ is a set of tuple constraints, where a *tuple constraint* can take four possible forms:

1. <var> <op> a
2. <var> $\in S$, <var> $\notin S$
3. <var> <op> <var>
4. <var> <op> <var> + b

where <var> is a variable in the scope of the smart table constraint, a and b are constants, S is a set of constants, and <op> is an operator on the set $\{<, \leq, =, \neq, \geq, >\}$.

The semantics of smart table constraints is simple and natural: a classical tuple τ is allowed by a smart table constraint sc iff there exists at least one smart tuple $\sigma \in table(sc)$ such that τ satisfies σ. A variable can be constrained multiple times. Note that when a variable $x \in scope(sc)$ is not involved in any tuple constraint of a smart tuple $\sigma \in table(sc)$, then x can take any value in its domain: such a variable is said to be *unrestricted* on σ and the set of unrestricted variables on σ is denoted by $unres(\sigma)$. Note also that any classical tuple $(a_1, \ldots, a_r)$ on a set of variables $\{x_1, \ldots, x_r\}$ can be re-written as a smart tuple: $\{x_1 = a_1, \ldots, x_r = a_r\}$.

As seen in the Introduction, smart tuples can help model constraints in a compact and natural way, when disjunction is needed. Smart table constraints can also be used to encode some global constraints. The encodings of *Lex*, *Max*, *Element* and *NotAllEqual* are smart table constraint versions of the ones proposed in [BW05]. In the examples below, tuple constraints are written directly inside the tables to ease reading. A tuple constraint of the form x_i <op> a (resp. x_i <op> $x_j + b$) is written as <op> a (resp. <op> $x_j + b$) in the column of the table corresponding to x_i. The following global constraints illustrate the modeling power of the smart table constraint. Their equivalent representation with classical tuples are exponentially larger. For instance, in the table for the element constraint, each smart tuple corresponds to d^m classical tuples. For compressed tuples, if only one variable is the target of all the tuple constraints, each smart tuple can be translated as d compressed tuples. This is the case for all the global constraints presented below except for *Lex*. For this constraint, the smart table is $O(d^m)$ times smaller than the table using compressed tuples. Short supports applied to table constraint can only efficiently encode unrestricted variables, making the encoding of each smart tuple $O(d^m)$ tuples with short supports for *Lex*, *Max*, *Min*, *Exactly1* and *AtMost1*. Global constraints are of course not the sole purpose of the smart table constraints but being able to efficiently encode those constraints has many advantages.

Lex($[x_1, \ldots, x_m], [y_1, \ldots, y_m]$***):*** $\bar{\boldsymbol{x}} > \bar{\boldsymbol{y}}$

x_1	x_2	$\ldots$	x_m	y_1	y_2	$\ldots$	y_m
$> y_1$	$*$	$\ldots$	$*$	$*$	$*$	$\ldots$	$*$
$= y_1$	$> y_2$	$\ldots$	$*$	$*$	$*$	$\ldots$	$*$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
$= y_1$	$= y_2$	$\ldots$	$> y_m$	$*$	$*$	$\ldots$	$*$

Max $([x_1, x_2, \ldots, x_m], M)$: $max(\bar{\mathbf{x}}) = M$

x_1	x_2	$\ldots$	x_m	M
$*$	$\leq x_1$	$\ldots$	$\leq x_1$	$= x_1$
$\leq x_2$	$*$	$\ldots$	$\leq x_2$	$= x_2$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
$\leq x_m$	$\leq x_m$	$\ldots$	$*$	$= x_m$

Min($[x_1, x_2, \ldots, x_m], M$***):*** $min(\bar{\boldsymbol{x}}) = M$

x_1	x_2	$\ldots$	x_m	M
$= M$	$\geq M$	$\ldots$	$\geq M$	$*$
$\geq M$	$= M$	$\ldots$	$\geq M$	$*$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
$\geq M$	$\geq M$	$\ldots$	$= M$	$*$

Element($I, [x_1, x_2, \ldots, x_m], R$***):*** $X[I] = R$

I	x_1	x_2	$\ldots$	x_m	R
$= 1$	$*$	$*$	$\ldots$	$*$	$= x_1$
$= 2$	$*$	$*$	$\ldots$	$*$	$= x_2$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
$= m$	$*$	$*$	$\ldots$	$*$	$= x_m$

Exactly1 $([x_1, x_2, \ldots, x_m], Y)$: $\#\{1 \leq i \leq m | x_i = Y\} = 1$

x_1	x_2	$\ldots$	x_m	Y
$= Y$	$\neq Y$	$\ldots$	$\neq Y$	$*$
$\neq Y$	$= Y$	$\ldots$	$\neq Y$	$*$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
$\neq Y$	$\neq Y$	$\ldots$	$= Y$	$*$

AtLeast1 $([x_1, x_2, \ldots, x_m], Y)$: $\#\{1 \leq i \leq m | x_i = Y\} \geq 1$

x_1	x_2	$\ldots$	x_m	Y
$= Y$	$*$	$\ldots$	$*$	$*$
$*$	$= Y$	$\ldots$	$*$	$*$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
$*$	$*$	$\ldots$	$= Y$	$*$

AtMost1 $([x_1, \ldots, x_m], Y)$: $\#\{1 \leq i \leq m | x_i = Y\} \leq 1$

x_1	x_2	$\ldots$	x_m	Y
$*$	$\neq Y$	$\ldots$	$\neq Y$	$*$
$\neq Y$	$*$	$\ldots$	$\neq Y$	$*$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
$\neq Y$	$\neq Y$	$\ldots$	$*$	$*$

NotAllEqual$(x_1, \ldots, x_m)$**:** $\exists 1 \leq i, j \leq m : x_i \neq x_j$

x_1	x_2	x_3	$\ldots$	x_m
$*$	$\neq x_1$	$*$	$\ldots$	$*$
$*$	$*$	$\neq x_1$	$\ldots$	$*$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
$*$	$*$	$*$	$\ldots$	$\neq x_1$

VectorDiff$([x_1, \ldots, x_m], [y_1, \ldots, y_m])$**:** $\bar{\boldsymbol{x}} \neq \bar{\boldsymbol{y}}$

x_1	x_2	$\ldots$	x_m	y_1	y_2	$\ldots$	y_m
$*$	$*$	$\ldots$	$*$	$\neq x_1$	$*$	$\ldots$	$*$
$*$	$*$	$\ldots$	$*$	$*$	$\neq x_2$	$\ldots$	$*$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
$*$	$*$	$\ldots$	$*$	$*$	$*$	$\ldots$	$\neq x_m$

Diffn$([x_1, \ldots, x_m], [i_1, \ldots, i_m], [y_1, \ldots, y_m], [j_1, \ldots, j_m])$**:**
no overlap between orthotopes defined in $\mathbb{R}^m$ from points $\bar{\mathbf{x}}$ and $\bar{\mathbf{y}}$ with lengths along axes of $\bar{\mathbf{i}}$ and $\bar{\mathbf{j}}$ respectively.

x_1	x_2	$\ldots$	x_m	y_1	y_2	$\ldots$	y_m
$*$	$*$	$\ldots$	$*$	$\geq x_1 + i_1$	$*$	$\ldots$	$*$
$\geq y_1 + j_1$	$*$	$\ldots$	$*$	$*$	$*$	$\ldots$	$*$
$*$	$*$	$\ldots$	$*$	$*$	$\geq x_2 + i_2$	$\ldots$	$*$
$*$	$\geq y_2 + j_2$	$\ldots$	$*$	$*$	$*$	$\ldots$	$*$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
$*$	$*$	$\ldots$	$*$	$*$	$*$	$\ldots$	$\geq x_m + i_m$
$*$	$*$	$\ldots$	$\geq y_m + j_m$	$*$	$*$	$\ldots$	$*$

5.2 FILTERING SMART TABLE CONSTRAINTS

This section presents a filtering algorithm to establish GAC on smart table constraints. As explained in Section 2.1, GAC is a property that relies on the concept of support. In general, identifying the set of supports of a constraint allows us to enforce GAC. All literals appearing in the set of supports are supported and thus GAC. This is the technique we will use to filter smart table constraints. The set of supports for a classical table constraint is the set of valid tuples. For smart table constraints, the set of supports is the union of the sets of supports of each smart tuple. Indeed, a single smart tuple can represent several classical tuples. Actually, for any smart table constraint sc, each smart tuple σ corresponds to a small CSP $P_\sigma = (X_\sigma, C_\sigma)$, with $X_\sigma = scope(sc)$ and $C_\sigma = \sigma$. The classical tuples that are supports of sc from σ are exactly the solutions in $sols(P_\sigma)$. Hence, the full set of supports of sc is equal to $\bigcup_{\sigma \in table(sc)} sols(P_\sigma)$. This is similar to the way sets of supports are computed for constructive disjunction.

Our objective is to efficiently identify and remove valid literals of sc without any support. It may seem costly to compute $sols(P_\sigma)$ for every smart tuple σ. Obtaining the set of supports for an arbitrary logical combination of constraints is NP-hard [BW05]. However, we impose that the constraint graph of any CSP P_σ that is associated with a smart tuple σ, is acyclic. We also have that the constraints in P_σ form one conjunction, as all the constraints of a CSP have to be respected. This restriction allows an efficient processing of the smart tuples when used for filtering.

Property 1. *Let σ be a smart tuple of a smart table constraint. P'_σ, the GAC closure of P_σ, is globally consistent, i.e., each literal of P'_σ appears in at least one solution of P_σ.*

This property is derived from [MF85] and the acyclic nature of the constraint graphs defined by smart tuples. This means that the set of literals in $sols(P_\sigma)$ is the set of generalized arc consistent literals of P_σ.

Obtaining the GAC closure on each of the P_σ and taking their union at the end to have the set of supported literals corresponds exactly to an application of the filtering rules defined in [BW05], when seeing the smart table constraint as a logical combination of basic arithmetic constraints. The acyclic nature of the conjunctions in the smart tuples guarantees that the set of supported literals computed by this procedure is the set of GAC literals for the logical combination of arithmetic constraints by Theorem 3 in [BW05]. Hence, this procedure is correct and computes the GAC literals for the smart table constraint.

Moreover, the complexity of filtering P_σ can also benefit from the form of the smart tuples, as expressed below.

Property 2. *The GAC closure of an acyclic binary CSP with e constraints can be obtained in $O(e \cdot F)$, where filtering an individual constraint if $O(F)$.*

The procedure for obtaining the GAC closure of an acyclic binary CSP $P = (X, C)$ is the following. As the CSP is acyclic, its constraint network forms a set of trees. Each tree is a connected component of the constraint network of the CSP. Each tree corresponds to an independent CSP, as two trees do not share variables (different connected components). We will call the set of CSPs resulting from the decomposition of a CSP P the forest of P (as it is a set of trees). Each CSP in a forest can be filtered independently since no variable is shared between them. For each tree T in a forest, revising constraints in turn from the deepest ones to the shallowest ones, and then the other way around, achieves GAC on T, thanks to its tree shape. Each constraint in C is thus revised two times and no fixed point needed. Revising a constraint c consists in removing the literals that have no support on c. We call this procedure GAC_tree. GAC_tree, as well as properties 1 and 2, are not original to this work, but they justify the filtering procedure of the smart table constraints.

Applying GAC_tree to a smart tuple σ of a constraint sc requires decomposing σ according to its connected components: the result of this decomposition will be denoted by forest(σ). More precisely, for each subset $cc \subseteq \sigma$ that represents a connected component, there is an associated tree T in forest(σ) that defines an independent sub-CSP (X_T, C_T) with $X_T = vars(cc)$ and $C_T = cc$. We shall refer to such sub-CSPs with tree shape as *treeCSPs*. An additional *void* tree T defining a trivial sub-CSP (X_T, C_T) with $X_T = unres(\sigma)$ and $C_T = \emptyset$ is introduced if $unres(\sigma) \neq \emptyset$. This guarantees that $sols(P_\sigma) = \Pi_{T \in trees(\sigma)} sols(T)$, which results from the independence of the trees w.r.t. each other.

The filtering algorithm proposed for smart table constraints, called smartSTR, works with the decompositions into treeCSPs instead of working directly with the smart tuples. It is inspired by STR (Simple Tabular Reduction) [Ull07, Lec11]. STR works by scanning constraint tables, going through each tuple sequentially. The validity of each row is checked. When a row is not valid, it is removed from the table. Otherwise, all the literals of the row are marked as having a support. After scanning the whole table, all the literals for which no support has been found are removed. The difference between STR and smartSTR is the way the validity checks and the collection of supported literals are performed. A smart tuple σ is valid iff P_σ admits at least one solution. A smart tuple σ is thus valid iff each treeCSP in forest(σ) admits at least

one solution. The literals supported by σ are the literals in $sols(P_\sigma)$ (obtained with GAC_tree). The supported literal set is then the union of the supported literal sets of each individual treeCSP in forest(σ).

Algorithm 13 presents the pseudo-code of smartSTR. It uses a data structure *sl* that contains all the literals without any found support (sl stands for supportless). Line 3 initializes *sl* with all valid literals (no support has been found yet). Then the algorithm loops over all the smart tuples of the constraint (line 4). The test at line 5 checks the validity of the current smart tuple by testing the validity of all its treeCSPs. If the smart tuple σ is valid, each of its independent treeCSP removes from *sl* the literals they support (loop at lines 6–7). The loop at line 8 empties the sets *sl* of all unrestricted variables on σ, as there is no restriction on those variables (actually, this corresponds to dealing with the void tree that is not in practice included in forest(σ)). If the smart tuple is invalid, it is removed from the table (line 9). The table has to be maintained during the search as it depends on the current domains. To have an efficient removal and restoration procedure, the table of the constraint is encoded with a sparse set data structure, as in STR1 and STR2. After going through all the smart tuples of the constraint, smartSTR removes the literals that are still left without a support (loop at line 10).

```
1  smartSTR(SmartTableConstraint sc):
2      // post: the constraint sc is GAC
3      forall x ∈ scope(sc): sl(sc)[x] ← D(x)
4      forall σ ∈ table(sc):
5          if (∧_{T∈forest(σ)} T.isValid()):
6              forall T ∈ forest(σ):
7                  T.collect(sl(sc))
8              forall x ∈ unres(σ): sl(sc)[x] ← ∅
9          else: remove σ from table(sc)
10     forall x ∈ scope(sc):
11         D(x) ← D(x) \ sl(sc)[x]
```

Algorithm 13: smartSTR

As seen in Algorithm 13, each treeCSP is responsible for checking its validity and removing from *sl* the literals it supports. This is done through the methods `isValid` and `collect`. Their specifications can be found in Interface 1, called TreeCSP. Note that a treeCSP involves a set of variables *vars* and belongs to the forest of a smart tuple σ.

From now on, the treeCSPs that are composed of only one constraint will be called *branches*. In the code presented below, we have specific classes for

```
interface TreeCSP
  fields: Variable[] vars
  isValid()
    // post: returns true iff the treeCSP is valid
  collect(Set{Value}[] sl)
    // pre:  the smart tuple σ, such that the treeCSP
    //       is in forest(σ), is valid
    // post: ∀x ∈ vars, ∀a ∈ D(x), (x,a) has a
    //       support in the treeCSP ⇒ a ∉ sl[x]
```

Interface 1: Interface for treeCSPs

unary branches (containing a tuple constraint of the form <var> <op> a, or <var> $\in S$), and *binary* branches (containing a tuple constraint of the form <var> <op> <var> , or <var> <op> <var> + b). There is one unary and binary branch class for each value of <op> . We also introduce one class for *simple* trees (trees of height 1 consisting of multiple branches all sharing the same root variable) and another one for *general* trees (trees of height > 1).

Algorithms 14, 15, 16 and 17 present the classes for unary branches with operations $=$, $\neq$, $<$ and $\in$ respectively. The pseudo-code for the other operators is very similar. The additional method `filterX`, not contained in Interface 1, is responsible for filtering the (pseudo) domain Dx given as argument. It is used by simple and general trees, where GAC has to be enforced on several branches. Dx is used to avoid filtering directly $\mathrm{D}(x)$, because an effective filtering can only be done when all smart tuples have been processed.

```
class UnaryBranchEq:TreeCSP /* x = a */
    fields: Variable x, Value a
    isValid(): return a ∈ D(x)
    collect(sl): sl[x] ← sl[x] \ {a}
    filterX(Dx): Dx ← Dx ∩ {a}
```

Algorithm 14: Classes for unary branche $=$

Algorithms 18, 19 and 20 present the classes introduced for binary branches with operations $=$, $\neq$ and $<$, respectively. Again, the pseudocode for the other operators is very similar. In their pseudocode, $S \oplus b$, where S is a set and b a value, represents the addition of the constant to all the values in the set. The definition of $\ominus$ is similar. When the domain of a variable x contains only one value, $\mathrm{D}(x)$ is considered as a value, and classical operators $+$ and $-$ are applied. They all implement the method `filterX`, as unary branches do,

```
1 class UnaryBranchNe:TreeCSP /* x ≠ a */
2     fields: Variable x, Value a
3     isValid(): return #D(x) > 1 ∨ a ∉ D(x)
4     collect(sl): sl[x] ← sl[x] ∩ {a}
5     filterX(Dx): Dx ← Dx \ {a}
```

Algorithm 15: Classes for unary branche $\neq$

```
1 class UnaryBranchLt:TreeCSP /* x < a */
2     fields: Variable x, Value a
3     isValid(): return min(D(x)) < a
4     collect(sl):
5         sl[x] ← sl[x] \ {b ∈ D(x) : b < a}
6     filterX(Dx): Dx ← Dx \ {b ∈ Dx : b ≥ a}
```

Algorithm 16: Classes for unary branche $<$

```
1 class UnaryBranchIn:TreeCSP /* x ∈ S */
2     fields: Variable x, Set S
3     isValid(): return ∃a ∈ S : a ∈ D(x)
4     collect(sl):
5         sl[x] ← sl[x] \ S
6     filterX(Dx): Dx ← Dx ∩ S
```

Algorithm 17: Classes for unary branche $\in$

but with two parameters (Dx and Dy). Dx is the copy of the domain of x to filter and Dy is a domain for y to use to filter Dx. The second parameter is needed during the execution of GAC_tree to use an already filtered copy of the domain of y to filter the copy of $D(x)$. Again, the filtering of the real domains of the variables can only occur after all the smart tuples have been processed. Those classes also implement a `filterY` method which is the counterpart of `filterX` for y. They implement a method `collectY`, used by simple trees to collect values, but only for the second involved variable y with respect to a (pseudo) domain Dx, given as a parameter, for the first involved variable x. It is called after Dx, which is initially a copy of $\mathrm{D}(x)$, has been filtered through the entire simple or general tree.

```
1  class BinaryBranchEq:TreeCSP /* x = y + b */
2      fields: Variable x, y
3              Value b
4      isValid(): return D(x) ∩ D(y) ⊕ b ≠ ∅
5      collect(sl):
6          I ← D(x) ∩ D(y) ⊕ b
7          sl[x] ← sl[x] \ I
8          sl[y] ← sl[y] \ I ⊖ b
9      collectY(sl, Dx):
10         I ← Dx ∩ D(y) ⊕ b
11         sl[y] ← sl[y] \ I
12     filterX(Dx,Dy): Dx ← Dx ∩ Dy ⊕ b
13     filterY(Dx,Dy): Dy ← Dx ⊖ b ∩ Dy
```

Algorithm 18: Classes for binary branche $=$

Algorithm 21 gives the pseudo-code for simple trees where all involved branches share the same root variable x (see the assertion at line 4). Since we can change the order of the variables in binary branches ($x_1 < x_2 \rightarrow x_2 > x_1$, etc.), this is not a requirement on the form of the smart tuples. This is enforced at the creation of the smart tuple's trees. The validity test at line 5 starts by making a copy Dx of $\mathrm{D}(x)$. As Dx is a field of the algorithm, its value can be used in other methods. Then, Dx is filtered through all branches (loop starting at line 7). The unary branches are treated at lines 8–9 and the binary ones, at lines 10–11. For the binary branches, `filterX` is called with the full domain of y as argument for y. If Dx does not become empty, that means that the simple tree has at least one solution. The method `collect` at line 13 uses Dx (which has already been filtered by `isValid`). Since all values in Dx have a support in the simple tree, they are removed from $\mathrm{sl}[x]$ (line 14). The loop at line 15 goes

```
1  class BinaryBranchNe:TreeCSP /* x ≠ y + b*/
2      fields: Variable x, y
3              Value b
4      isValid():return #D(x) > 1 ∨ #D(y) > 1
5                       ∨ D(x) ≠ D(y) + b
6      collect(sl):
7          if #D(x) > 1: sl[y] ← ∅
8          else: sl[y] ← sl[y] ∩ D(x) − b
9          if #D(y) > 1: sl[x] ← ∅
10         else: sl[x] ← sl[x] ∩ D(y) + b
11     collectY(sl, Dx):
12         if #Dx > 1: sl[y] ← ∅
13         else: sl[y] ← sl[y] ∩ Dx − b
14     filterX(Dx,Dy):
15         if #Dy = 1: Dx ← Dx \ Dy + b
16     filterY(Dx,Dy):
17         if #Dx = 1: Dy ← Dy \ Dx − b
```

Algorithm 19: Classes for binary branche $\neq$

```
1  class BinaryBranchLt:TreeCSP /* x < y + b */
2      fields: Variable x, y
3              Value: b
4      isValid(): return min(D(x)) < max(D(y)) + b
5      collect(sl):
6          sl[x] ← sl[x] \ {a ∈ D(x) : a < max(D(y)) + b}
7          sl[y] ← sl[y] \ {c ∈ D(y) : c > min(D(x)) − b}
8      collectY(sl, Dx):
9          sl[y] ← sl[y] \ {c ∈ D(y) : c > min(Dx) − b}
10     filterX(Dx,Dy):
11         Dx ← Dx \ {a ∈ Dx : a ≥ max(Dy) + b}
12     filterY(Dx,Dy):
13         Dy ← Dy \ {c ∈ Dy : c ≤ min(Dx) − b}
```

Algorithm 20: Classes for binary branches $<$

through every binary branch (i.e., with a scope containing two variables) to collect the supported values for the second involved variables (y) from the filtered domain Dx. The supported values for the variables y are directly removed from sl instead of copying their domain and filtering it. Note that `isValid` and `collect` are adaptations of the two-pass filtering GAC_tree. During the first pass, only the domain of x (actually, Dx) is filtered. Indeed, as it may change at each new processed branch, filtering the domains of variables y (actually, updating sl) is useless at that time. The validity test is not concerned in the second pass because if x still has values in its domain after the first pass, the simple tree is guaranteed to have at least one solution.

```
1  class SimpleTree:TreeCSP
2      fields: Variable x, TreeCSP[] branches,
3              Domain Dx
4      assert ∀T ∈ branches : T.x = x
5      isValid():
6          Dx ← D(x)
7          forall T ∈ branches:
8              if #T.vars = 1
9                  T.filterX(Dx)
10             else
11                 T.filterX(Dx,D(y))
12         return Dx ≠ ∅
13     collect(sl):
14         sl[x] ← sl[x] \ Dx
15         forall T ∈ branches : #T.vars = 2
16             T.collectY(sl,Dx)
```

Algorithm 21: Class for simple trees

The class for general trees is given in Algorithm 22. This algorithm uses several fields. The array `allVars` contains all the variables appearing in the tree. The two-dimensional array `branches` contains all the branches for each level of the tree, from 1 (branches containing the root variable) to `treeHeight`. The array `domCopy` contains the copies of the domains of the variables of the tree that are used during the procedure GAC_tree. For this algorithm, we will suppose that for all the binary branches, the variable x is always the closest to the root. This is again enforced during the creation of the smart tuple's trees. The assertion of line 5 thus checks that all the variables x have a corresponding variable as y at the level below (closer to the root). The `isValid` method (line 7) realizes the first pass of GAC_tree (using copies of the domains), filtering the domains of the different x variables from the leaves to the root. If the

variable at the root (branches$[1][1].x$) of the tree still has values, it returns true. Its `collect` method (line 17) then carries out the second pass by filtering the (copies of the) domains of the y variables of the branches. It also removes supported values from sl. At this point, it is important to note that the code presented for unary branches, binary branches, and simple trees already covers all the examples given in this chapter.

```
1  class GeneralTree:TreeCSP
2      fields: Variable[] allVars, TreeCSP[][] branches,
3                Value treeHeight, Domain[] domCopy
4      assert ∀1 < l ≤ treeHeight, ∀b ∈ branches[l],
5                ∃b2 ∈ branches[l − 1] : b.x = b2.y
6      isValid():
7          forall x ∈ allVars :
8              domCopy[x] ← D(x)
9          forall l ∈ treeHeight..1 :
10             forall T ∈ branches[l]:
11                 if #T.vars = 1
12                     T.filterX(domCopy[T.x])
13                 else
14                     T.filterX(domCopy[T.x], domCopy[T.y])
15         return domCopy[branches[1][1].x] ≠ ∅
16     collect(sl):
17         forall l ∈ 1..treeHeight :
18             forall T ∈ branches[l]:
19                 sl[T.x] ← sl[T.x] \ domCopy[T.x]
20                 if #T.vars = 2:
21                     T.filterY(domCopy[T.x], domCopy[T.y])
22         forall T ∈ branches[treeHeight] : #T.vars = 2
23             sl[T.y] ← sl[T.y] \ domCopy[T.y]
```

Algorithm 22: Class for general trees

We now study the complexity of our approach. The complexity of filtering a smart tuple depends on the complexity of filtering each of its treeCSPs, as they are independent. For a smart tuple σ (on variables with maximal domain size d), the time complexities for the different operators are

- for the unary branches:

<op>	`isValid`	`collect`	`filterX`
$=$	$O(1)$	$O(1)$	$O(1)$
$\neq$	$O(1)$	$O(1)$	$O(1)$
$>\geq<\leq$	$O(1)$	$O(d)$	$O(d)$
$\in$	$O(d)$	$O(d)$	$O(d)$

- for the binary branches:

<op>	`isValid`	`collect/` `collectY`	`filterX/` `filterY`
$=$	$O(d)$	$O(d)$	$O(d)$
$\neq$	$O(1)$	$O(1)$	$O(1)$
$>\geq<\leq$	$O(1)$	$O(d)$	$O(d)$

Each tuple constraint is either its own tree or belongs to a larger tree. If the branch is its own tree, the time complexities of `isValid` and `collect` are $O(d)$ for any operator. If the branch is included in a simple or general tree, then GAC_tree guarantees that the `collectY`, `filterX` and `filterY` methods are called a constant number of times. The time spent testing the validity and performing value collection in one branch is thus $O(d)$. This makes the treatment of one smart tuple with k tuple constraints $O(k \cdot d + r)$, where r is the arity of the table constraint. The last term comes from the treatment of unrestricted variables. Initializing sl at the beginning of smartSTR and removing the unsupported values from the domains at the end are $O(r \cdot d)$. The total time complexity of one call to smartSTR for a smart table constraint of arity r with t smart tuples is thus $O(r \cdot d + t \cdot k \cdot d + t \cdot r)$. For a classical table constraint of arity r with t' tuples, STR2 then has a time complexity of $O(r \cdot d + t' \cdot r)$. In all the examples given, $k \leq r$ (less tuple constraints than variables). Also, the number of smart tuples is at least $d + 1$ times less than the number of classical tuples. In these conditions, the complexity of filtering the smart table is less than the complexity of using STR2 on the table without smart tuples. Indeed, $t \cdot k \cdot d + t \cdot r \leq t' \cdot r$.

5.3 LINK WITH LOGICAL COMBINATION OF CONSTRAINTS

A table constraint can be viewed as a disjunction of tuples. Indeed, to satisfy a table constraint, its variables have to take values corresponding to one of the tuples of the table. Smart constraints can also be viewed as large disjunction. The difference is that smart tuples are conjunctions of basic arithmetic constraints. Smart table constraints can thus be viewed as large disjunctions of conjunctions of basic arithmetic constraints. Indeed, each smart tuple contains a conjunction of basic arithmetic constraints and the table is a disjunction of such tuples, since the variables can satisfy any of the smart tuples. The filtering of logical combinations of constraints has already been studied in the literature [CC95, WM96, VHSD98, BR98, KB01, Lho04, BW05, JMNP10, Lho12]. Filtering general logical combinations of constraints is NP-Hard in general. However, smart table constraints are particular kinds of logical combinations. In this chapter, we showed how Simple Tabular Reduction (STR) [Ull07, Lec11] can be adapted for smart table constraints to produce an efficient filtering procedure to enforce GAC in polynomial time. Smart table constraints encode a subset of the logic algebra defined in [BW05]: the form of the logical combinations (disjunction of conjunction, conjunctions forming acyclic networks) is a subset of the logical combinations defined in [BW05] and we restrict the constraints that can be combined to be simple arithmetic constraints. The rules for the filtering are however directly derived from [BW05]. While restricting the expressive power compared to [BW05], smart tuples greatly increase the expressive power over classical tuples. More importantly, those restrictions allows our propagator, smartSTR, to efficiently compute the GAC maximal inconsistent set of the constraint, using the GAC maximal inconsistent sets of all the smart tuples. GAC maximal inconsistent sets cannot be guaranteed in general for the full logic algebra from [BW05]. The novelty in the present approach lies in the introduction of a concrete propagator for such a subset of the logic algebra. These restrictions also allow the filtering of smart tuples to be much simpler than the filtering for the general conjunctions defined in [BR98, KB01, Lho04]. The pruning achieved by the disjunction is equivalent to the pruning of constructive disjunction [WM96, VHSD98]. The propagation of a whole table constraint can even be seen as the propagation of a large constructive disjunction. The ability to leave variables out of the constraints in the smart tuple makes their filtering as efficient as the improved constructive disjunction filtering defined in [Lho04]. In [CC95], the authors proposed a filtering for constructive disjunction based on indexicals as well as a stronger filtering, considering disjunctions together with other constraints. In this chapter, we do not investigate propagating more than one table constraint at a time.

The filtering for both conjunctions and disjunctions proposed in this chapter is stronger than the light filtering proposed in [JMNP10], thanks to the hypothesis made on the form of the smart tuples.

5.4 EXPERIMENTAL RESULTS

The optimizations present in STR2 can also be included in smartSTR. The obtained algorithm is then called smartSTR2. Comparing SmartSTR2 with all the specialized algorithms developed over the years for the global constraints mentioned earlier is clearly beyond the scope of this chapter. However, we shall show the interest of SmartSTR2 with a few case studies. Comparing a propagator F with SmartSTR2 on a global constraint means that in the same CSP, all the instances of the global constraint are either propagated with F or their encoding in a smart table constraint is propagated with SmartSTR2. We conducted experiments (with the solver AbsCon) on a laptop computer, equipped with Intel(R) Core(TM) i7-2820QM CPU @ 2.30GHz, under Linux. The results are given in seconds, or correspond to the number of visited nodes per second. We have checked that all the tested approaches were traversing the exact same search trees (most of the time using dom/ddeg as the variable ordering heuristic for this purpose). As the algorithms compared are different from one instance set to one another, we will present the results of the statistical treatment of the experimental data (single class setting) for each of the instance sets separately.

TAL instances In natural language processing, one possible task is to measure how well-formed a sentence is (i.e., to what extent it respects a grammar). A constraint model (R. Coletta, personal communication) has been recently developed for this problem, denoted here by TAL. It involves the Element constraint (with R as a variable, as described earlier in the chapter). Instances for this optimization problem are defined by entering an input sentence. In this model, Element constraints represent about 8% of the constraints. We compare SmartSTR2 with GACElt, which corresponds to the GAC propagator based on watched literals [GJM06]. In this context, the two algorithms show very close performances as shown by Table 5.1. Figure 5.1 plots the percent bests for this instance set (and also for the BIBD instance set). The percent best of one algorithm is the percentage of the mean time taken by the algorithm to solve the instances with respect to the mean time taken by the fastest algorithm. For the statistical treatment of those results we used the θ_1 statistic of interest (arithmetic mean of differences) with 10,000 bootstrap sets and a confidence level α of 0.05, since no timeouts were present in the experimental data. The con-

sentence	GACElt	SmartSTR2
phrase1	3.6	3.7
phrase2	17.6	17.9
phrase3	54.4	54.2
phrase4	46.8	46.8
phrase5	82.4	82.6

Table 5.1: CPU time to solve *TAL* instances.

fidence interval for $\theta_1(\text{SmartSTR2}, \text{GACElt})$ is [-0.24, 0.06] (positive values correspond to SmartSTR2 being faster). This indicates that no propagator is significantly faster than the other on those instances, as 0 is inside the (very small) confidence interval.

BIBD instances A *BIBD* is a standard combinatorial problem. We consider here the model introduced in [MT99] and the series of instances tested in [FHK+02]. There is a lexicographic constraint between any two adjacent rows or columns. We compare SmartSTR2 with GACLex, which corresponds to the filtering procedure described in [Lec09] and is a variant of [FHK+06]. Table 5.2 shows the results we have obtained with both algorithms. Figure 5.1 plots the percent bests for this instance set. Interestingly, one can observe that using SmartSTR2 instead of GACLex has a very limited cost. This is interesting because SmartSTR2 is general-purpose while GACLex is a specialized propagator. Similar results are obtained with the social golfer problem. For the statistical treatment of the results, as for the previous instance set, we used the θ_1 statistic of interest with the same parameters. The confidence interval for $\theta_1(\text{SmartSTR2}, \text{GACLex})$ is [-4.4, -0.77], indicating that GACLex is significantly faster than SmartSTR2 on those instances but only slightly (the values in the confidence intervals correspond to seconds).

Rectangle Packing instances The *RectanglePacking* problem [SO08] consists of placing all squares from size 1×1 to $n \times n$ into a rectangle of size $w \times h$ without overlap between the squares. We adopt the model and search parameters given in [NGJM13, JN13]. Table 5.3 shows the nodes searched per seconds within a given time limit, as the problems are too complicated to be

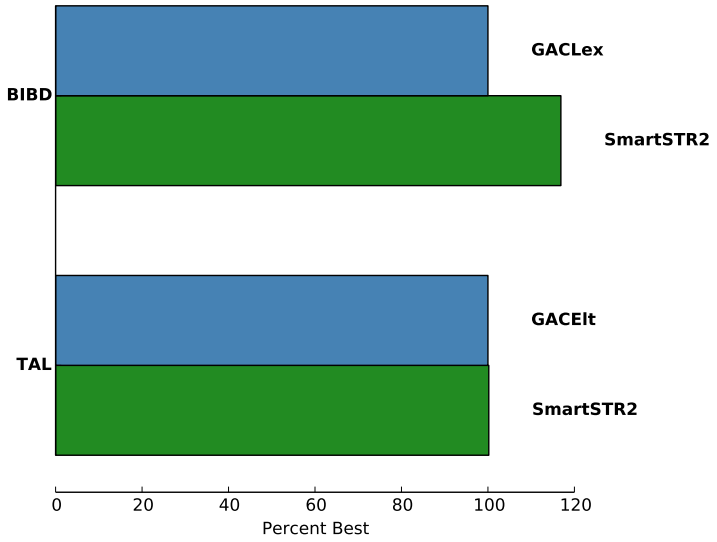


Figure 5.1: Percent best quantities for the TAL and BIBD instance sets

v-b-r-k-λ	GACLex	SmartSTR2
6-50-25-3-10	1.3	1.6
6-60-30-3-12	1.5	2.1
6-70-35-3-10	2.2	2.8
10-90-27-3-6	5.8	7.3
9-108-36-3-9	11.4	14.2
15-70-14-3-2	7.4	7.9
12-88-22-3-4	7.0	8.3
9-120-40-3-10	17.9	25.1
10-120-36-3-8	10.6	14.0
13-104-24-3-4	99.1	108.6

Table 5.2: CPU time to solve *BIBD* instances.

n-w-h	GAC-valid	ShortSTR2	SmartSTR2
18-31-69	1,821	2,784	57,249
19-47-53	2,003	3,166	57,221
20-34-85	1,324	1,579	45,600
21-38-88	849	1,295	40,600
22-39-88	981	1,035	41,162
23-64-68	983	1,292	40,495
24-56-88	446	790	32,758
25-43-129	661	347	30,544
26-70-89	544	703	31,374
27-47-148	326	175	26,786

Table 5.3: Nodes searched per second for *RectanglePacking* instances.

solved completely. Figure 5.2 plots the percent bests for this instance set (and also for the AllDistinctVector instance set). In this context of nodes searched per second, the percent best of algorithm A is the percentage of the best algorithm with respect to A. These results suggest that SmartSTR2 is very efficient on this problem. It clearly outperforms ShortSTR2, and seems to be at least as efficient as the other methods proposed in [NGJM13] (not implemented in our system) when we compare their results with ours. Note that GAC-valid (sometimes called GAC-schema) is another general approach, given here as a baseline. For the statistical treatment of the experimental data, no timeout was enforced, and hence we used θ_1 with the same configuration as for the previous instance sets. However, as a larger node count per second is better, we aggregate the difference between the first algorithm's node count and the second. This is the opposite of what was done for time differences. This was done so as to make a positive value indicate that the first algorithm is faster. The confidence interval $\theta_1(\text{SmartSTR2}, \text{GAV-valid})$ is [33308.5, 45122.6] and the one for $\theta_1(\text{SmartSTR2}, \text{ShortSTR2})$ is [33157.4, 44445.8]. The values corresponds to nodes searched per second. This indicates that SmartSTR2 is significantly faster than both GAC-valid and ShortSTR2 on those instances.

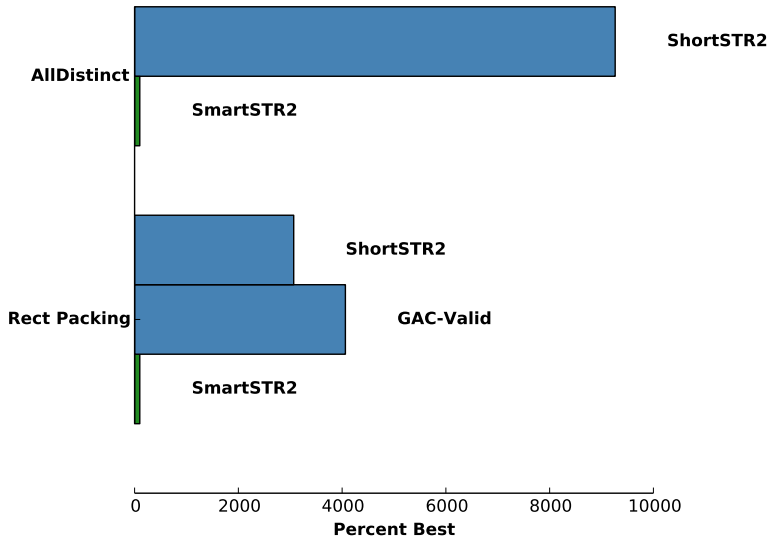


Figure 5.2: Percent best quantities for the Rectangle Packing and AllDistinctVectors instance sets

AllDistinctVectors instances For our last experiment, we consider the Case Study 4 in [JN13], where a problem, denoted by *AllDistinctVectors* here, involves the VectorDiff constraint. An instance *p-a-d* of this problem has exactly p vectors (arrays of variables), each vector has length a and the domain of each variable has a size of d. For any pair of vectors, the two vectors must be distinct. In [JN13], it has been shown that ShortSTR2 is an interesting competitor to HaggisGAC. The results are given in Table 5.4 and the percent best quantities are plotted in Figure 5.2. For instances where all variables are Boolean (i.e., $d = 2$), SmartSTR2 is slightly slower than ShortSTR2 (because the tables are small). However, when we increase d, Table 5.4 shows that just when applying GAC stand-alone, SmartSTR2 is clearly faster than ShortSTR2. This can be explained by the size of the constraint tables. For example, for *40-100-40*, the tables contain $156,000$ and 100 tuples in ShortSTR2 and SmartSTR2, respectively. As for the previous instance sets, the statistical treatment of the experimental data is performed with the θ_1 statistic of interest with the same parameters. The confidence interval for $\theta_1(\text{SmartSTR2}, \text{ShortSTR2})$ is here [2.25, 27.3], indicating that SmartSTR2 is significantly faster than ShortSTR2 on those instances, but only slightly.

p-a-d	ShortSTR2	SmartSTR2
40-100-2	0.07	0.07
40-100-8	1.55	0.18
40-100-16	6.49	0.18
40-100-24	14.7	0.19
40-100-32	28.1	0.20
40-100-40	44.5	0.21

Table 5.4: CPU time to enforce GAC on *AllDistinctVectors* instances.

There is no global statistical treatment of the experimental data presented for this section. The reason is that the algorithms compared to SmartSTR2 are different for each dataset (except for ShortSTR2 on the last two datasets where the measurement is the number of nodes for the first and the time for the second), making the integration of all the datasets in one multi-class bootstrap test impossible.

6

EFFICIENT FILTERING PROCEDURE FOR DOMAIN k-WISE CONSISTENCY ON TABLE CONSTRAINTS

As we have seen in the previous chapters, propagation is an important concept in constraint programming. The target consistency of chapters 4 and 5 is generalized arc consistency. While this consistency provides the strongest pruning one can achieve by inspecting the constraints independently, the pruning obtained can be increased by considering several constraints at a time, which integrates the information of several constraints at once and hence performs a more global reasoning. The more global the reasoning, the better the pruning. One consistency considering several constraints at a time is called k-wise consistency (kWC) (see Section 2.3). In a nutshell, k-wise consistency considers all possible groups of k constraints. For each group, it ensures that the allowed tuples of the constraints are consistent with each other. Accepting a tuple that is inconsistent with the accepted tuples of a group of constraint is useless. kWC thus removes from the constraints the allowed tuples that cannot be extended to a larger tuple respecting all the constraints of the group. At the end, kWC ensures that when a tuple is accepted by a constraint, it will not be ruled out

later, when extended to any group of k constraints. The k-wise consistency filters the constraints, while GAC filters the domains of the variables. Filtering the constraints is another way to reduce the search space. K-wise consistency is incomparable to GAC: there are CSP that are k-wise consistent but not GAC, and vice versa. In this chapter, we combine k-wise consistency and GAC for table constraints. The restriction to table constraints is driven by their adequacy to k-wise consistency. Indeed, the explicit access to the list of tuples allowed by the constraints is particularly useful for k-wise consistency. The constraint filtering nature of k-wise consistency may render its integration into existing solvers difficult. The combination proposed in this chapter, already presented in Section 2.3, thus filters the domains of the variables without changing the constraints. It is called *domain k-wise consistency*, and is the focus of this chapter. Often, consistencies stronger than GAC require complex propagators to be enforced. In this chapter, we propose a filtering procedure for our strong consistency solely relying on existing GAC propagators for table constraints. We start by recalling the definition of domain k-wise consistency. Section 6.2 presents a filtering procedure for k-wise consistency that is useful for the filtering procedure of domain k-wise consistency, presented in Section 6.3. Section 6.4 then presents two variants of our filtering procedure, reducing their cost in practice. This chapter ends with experimental results of our filtering procedure. Again, the definitions and the filtering procedure are presented for CSPs but can be used in the context of constraint optimization problems.

Related Publication

> [MDL14] Jean-Baptiste Mairy, Yves Deville and Christophe Lecoutre, "Domain k-Wise Consistency Made as Simple as Generalized Arc Consistency" Integration of AI and OR Techniques in Constraint Programming (CPAIOR) 2014, pages 235-250

6.1 DOMAIN k-WISE CONSISTENCY

Combinations of generalized arc consistency with kWC have already been studied in the literature. Indeed, to integrate the filtering of the constraints into the domains of the variables, and hence speed the search up, GAC is the perfect consistency. Generalized arc consistency and 2-, 3- and k-wise consistency have been combined in [Jég91, JJNV89, BSW08, Ste08, KWR+10, Ste07]. One such combination is GAC+kWC (see Section 2.3), where a CSP is GAC+kWC iff it is both GAC and kWC. But, as kWC is constraint filtering, GAC+kWC is both domain and constraint filtering. Interestingly, from

GAC+kWC, we can derive our domain-filtering consistency: the *domain k-wise consistency*, or DkWC in short. Intuitively, for DkWC, all literals are required to have a k-wise consistent support (while for GAC, they are only required to have a support). When a CSP P is domain k-wise consistent, it means that all variable domains of P cannot be reduced when enforcing GAC+kWC. The formal definition of DkWC from Section 2.3 is given below.

Definition 12. (DkWC) *A CSP $P = (X, D, C)$ is* domain k-wise consistent *(DkWC) iff GAC+kWC(P) is a CSP $Q = (X, D^Q, C^Q)$ such that $D = D^Q$.*

The remainder of this chapter will be devoted to a filtering procedure for DkWC for table constraints using solely existing GAC propagators. The filtering is based on a filtering procedure for kWC, which is presented in the next section.

6.2 FILTERING PROCEDURE FOR k-WISE CONSISTENCY

The filtering procedure for DkWC, presented in the next section, is based on a filtering technique for kWC. This filtering technique for kWC is thus presented in this section. It is a technique that uses an alternative form of the CSP called the k-dual CSP. The k-dual CSP focuses on constraints where the CSP focuses on variables. Filtering the k-dual with GAC achieves kWC constraint filtering on the original CSP. This is the reason why we used this technique as a base for our DkWC filterign.

The filtering procedure for kWC is a generalization to the k-wise case of the filtering process presented in [JJNV89], and different from the one presented in [Jég91]. Due to explicit access to the list of allowed tuples, table constraints are particularly adapted for strong constraint-filtering consistencies. The filtering procedures proposed in this chapter are thus designed for such table constraints. From now on, until the end of the chapter, all constraints will be assumed to be table constraints.

As kWC is a constraint-filtering consistency, the idea is to define and use a special dual form of the given CSP in order to obtain kWC by enforcing GAC on the dual representation. Because this dual form depends on k, we call it *k-dual* CSP. This is a generalization of the dual used in [JJNV89] that is equivalent to the *order k constraint graph* defined in [Jég91].

Specifically, the k-dual of a CSP contains one *dual* variable x'_i per constraint c_i in the original CSP and one *k-dual* constraint c'_j per group of k original distinct constraints. Each variable x'_i has a domain which is the set of indexes of the tuples in the original constraint c_i, and each constraint c'_j is a table constraint representing the join of k original constraints. Note that the tuples in

those new tables are represented with the indexes of the tuples in the original constraints, which allows the new constraints to have arity k only.

Definition 13. (k-dual CSP) *Let $P = (X, D, C)$ be a CSP. The* k-dual *of P is the CSP $P^{kd} = (X^{kd}, D^{kd}, C^{kd})$ where:*

- *for each constraint $c_i \in C$, X^{kd} contains a variable x'_i with its domain defined as $D^{kd}(x'_i) = \{1, 2, \ldots, c_i.length\}$,*
- *for each subset S of k constraints of C, C^{kd} contains a constraint c' such that $scope(c') = \{x'_i \mid c_i \in S\}$ and c' is a k-ary table constraint containing the join of all constraints in S (represented with the indexes of the original tuples).*

If P^{kd} is the k-dual of P, then variables and constraints of P are said to be *original* whereas variables and constraints of P^{kd} are said to be *dual*. An example of a k-dual CSP for $k = 3$ can be found in Example 7.

Example 7. Let $P = (X, D, C)$ be a CSP such that $X = \{u, v, w, x, y, z\}$, $D = \{1, 2, 3, 4\}^6$ and $C = \{c_1, c_2, c_3\}$ where:

- $scope(c_1) = \{u, v, w\}$ and $table(c_1) = \{(1, 2, 3), (1, 2, 4)\}$,
- $scope(c_2) = \{u, x, y\}$ and $table(c_2) = \{(1, 3, 4), (2, 3, 4)\}$,
- $scope(c_3) = \{v, x, z\}$ and $table(c_3) = \{(2, 3, 1), (3, 3, 2)\}$.

The 3-dual of P is a CSP $P^{kd} = (X^{kd}, D^{kd}, C^{kd})$ such that $X^{kd} = \{x'_1, x'_2, x'_3\}$, $D^{kd} = \{1, 2\}^3$, and $C^{kd} = \{c'\}$ with $scope(c') = \{x'_1, x'_2, x'_3\}$ and $table(c') = \{(1, 1, 1), (2, 1, 1)\}$. It represents the full join of the original constraints on $\{u, v, w, x, y, z\}$, which is composed of the tuples $(1, 2, 3, 3, 4, 1)$ and $(1, 2, 4, 3, 4, 1)$. For example, the first tuple is obtained by joining the first tuple of c_1, the first tuple of c_2 and the first one of c_3.

Property 3. *A CSP is kWC iff its k-dual CSP is GAC. [JJNV89, Jég91].*

Property 3 was introduced in [JJNV89] for $k = 2$ and the general result was established in [Jég91] for a similar k-dual CSP. A filtering procedure to enforce GAC+2WC (i.e., both pairwise consistency and generalized arc consistency) on a CSP P consists in (1) enforcing GAC on the 2-dual of P, then (2) restraining the constraints of P in order to only contain tuples corresponding to valid dual values, and finally (3) establishing GAC on P [Jég91, BSW08]. The generalization of this procedure to the k-wise case uses the k-dual instead of the 2-dual.

6.3 DOMAIN k-WISE CONSISTENCY FILTERING

In this section, we propose to reformulate the CSP to be solved in order to be able to enforce DkWC in a single step, just by applying classical GAC filtering. Basically, to enforce DkWC solely with GAC propagators, we merge the CSP with its k-dual. GAC on the merged CSP ensures the DkWC of the original CSP. For a CSP P, the merge is done by adding to P all variables and all constraints from the k-dual CSP of P; the dual variables and the original constraints are then linked. Without this link, the removal of a value from a dual variable would not leverage its corresponding original constraint (and reciprocally). In the definition of GAC+kWC (on which DkWC is based), only valid tuples can serve as supports either for values (generalized arc consistency part) or for other tuples (k-wise consistency part). The link we make guarantees that original tuples corresponding to invalid dual values are invalidated, and reciprocally, ensuring that original constraints and dual variables keep the same pace during filtering. This ensures that the k-wise inconsistent tuples cannot be used as supports for values and that invalid tuples cannot be used as k-wise support for tuples. The link we propose is integrated directly into the constraints. Each original constraint c_i is thus transformed into a *hybrid* constraint $\phi(c_i)$ involving the original variables in the scope of c_i as well as the dual variable x'_i that is associated with c_i. For each tuple τ in $table(c_i)$, we generate a tuple in $table(\phi(c_i))$ by simply appending to τ its position in $table(c_i)$. In the following definition, if τ is a r-tuple $(a_1, \ldots, a_r)$ then $\tau \odot b$ denotes the $(r+1)$-tuple $(a_1, \ldots, a_r, b)$.

Definition 14. (Hybrid Constraints) *Let $P = (X, D, C)$ be a CSP. The set of hybrid constraints $\phi(C)$ of P is the set $\{\phi(c_i) \mid c_i \in C\}$ where:*

- $scope(\phi(c_i)) = scope(c_i) \cup \{x'_i\}$
- $table(\phi(c_i)) = \{\tau_j \odot j \mid \tau_j$ *is the* j^{th} *tuple of* $table(c_i)\}$

with x'_i denoting the dual variable associated with c_i.

In this way, the removal of a value j from $D(x'_i)$ will be reflected in $\phi(c_i)$, as the tuple $\tau_j \odot j$ will be invalidated. Also, when the tuple $\tau_j \odot j$ becomes invalid due to a value removed from the domain of an original variable, j will be removed from $D(x'_i)$ as $\tau_j \odot j$ is the only support of (x'_i, j) in $\phi(c_i)$. We can now introduce *k-interleaved CSPs*, containing both the original CSP with hybrid constraints and the k-dual CSP.

Definition 15. (k-Interleaved CSP) *Let $P = (X, D, C)$ be a CSP. The* k-interleaved *of P is the CSP $P^{ki} = (X^{ki}, D^{ki}, C^{ki}) = (X \cup X^{kd}, D \cup D^{kd}, \phi(C) \cup$

C^{kd}) *where* (X^{kd}, D^{kd}, C^{kd}) *is the k-dual of* P *and* $\phi(C)$ *the hybrid constraints of* P.

As a constraint optimization problem is a CSP with an objective function, the k-interleaved COP is defined similarly, with the objective unchanged. The properties defined hereafter on the k-interleaved CSP are trivially also valid for the k-interleaved COP.

The following property shows an interesting connection: enforcing GAC on the k-interleaved CSP of a CSP P is equivalent to enforcing GAC+kWC on P, when the focus is only on the domains of the variables of P.

Property 4. *Let* $P = (X, D, C)$ *be a CSP and* $P^{ki} = (X^{ki}, D^{ki}, C^{ki})$ *be the k-interleaved CSP of* P. *If* $Q = (X, D^Q, C^Q)$ *is the GAC+kWC-closure of* P *and* $R = (X^{ki}, D^R, C^{ki})$ *is the GAC-closure of* P^{ki}, *then we have* $D^Q = D^R[X]$ *(i.e.,* $D^Q(x) = D^R(x), \forall x \in X$*).*

Proof.

(a) $\forall x \in X : (x,a) \in D^R \Rightarrow (x,a) \in D^Q$:

From the definition of GAC applied to P^{ki}, we know that:

$$(x,a) \in D^R \Leftrightarrow$$
$$\forall c_i \in C, x \in scope(c_i) : \exists \tau \in table(\phi(c_i)),$$
$$\tau \in D^R(scope(\phi(c_i))) \wedge \tau[x] = a$$

We also have that:

$$\tau \in D^R(scope(\phi(c_i)))$$
$$\Rightarrow \forall x \in scope(c_i) : \tau[x] \text{ is GAC in } P^{ki}$$
$$\Rightarrow \forall x \in scope(c_i) : \tau[x] \text{ are GAC+KwC in } P \text{ because each support in } P^{ki} \text{ includes a k-dual variable, precisely encoding the k-wise consistency (Property 3).}$$

By Property 3, since $\tau[x'_i] \in D^R(x'_i)$ (x'_i being the dual variable of c_i), τ is kWC.

We thus have $\tau[scope(c_i)] \in D^Q(scope(c_i))$ and kWC, which implies $(x,a) \in D^Q$

(b)$\forall x \in X : (x,a) \in D^Q \Rightarrow (x,a) \in D^R$:

From the definition of GAC+kWC, we know that:

$$(x,a) \in D^Q \Leftrightarrow$$
$$\forall c_i \in C, x \in scope(c_i) : \exists \tau \in table(c_i^Q),$$
$$\tau \in D^Q(scope(c_i)), \tau[x] = a$$

where $c_i^Q \in C^Q$ is the filtered version of $c_i \in C$.

By Property 3, we have that

$$\tau \in table(c_i^Q) \Rightarrow (x_i', j) \in D^R(x_i')$$

where $x_i' \in X^{ki}$ is the dual variable of c_i and j the tuple index of τ.

We also have that

$$\tau \in D^Q(scope(c_i))$$
$$\Rightarrow \forall x \in scope(c_i) : \tau[x] \text{ has a valid kWC support in } P.$$
$$\Rightarrow \forall x \in scope(c_i) : \tau[x] \text{ is GAC in } P^{ki}$$

We thus have $\tau \oplus j \in D^R(scope(\phi(c_i)))$, which implies $(x,a) \in D^R$

□

The intuition of the proof is as follows. On the one hand, each literal (x,a) of $D^R[X]$ is supported on each constraint c^{ki} involving x by a valid tuple in R. As all supports on constraints of C^{ki} include a valid dual variable, we have that $(x,a) \in D^Q$. On the other hand, each literal (x,a) of D^Q is supported on each constraint c^Q involving x by a valid tuple in Q. This tuple is k-wise consistent in Q. By Property 3, the dual variables in X^{ki} precisely encode this k-wise consistency.

Then, we can deduce the following corollary.

Corollary 1. *If the k-interleaved CSP of a CSP P is GAC then P is DkWC.*

It is important to note that the relation between a CSP and its k-interleaved CSP is preserved through value refutation. In the following properties, for any value refutation $x \neq a$, $P|_{x \neq a}$ denotes the CSP P where the value a is removed from $D(x)$, and for any set of value refutations Δ, $P|_\Delta$ is defined similarly. The preservation of "k-interleavedness" is stated by the following property.

Property 5. *Let $P = (X, D, C)$ be a CSP and P^{ki} be the k-interleaved CSP of P. $\forall x \in X, \forall a \in D(x)$, $P^{ki}|_{x \neq a}$ is the k-interleaved CSP of $P|_{x \neq a}$.*

Proof. From the definition of k-interleaved CSPs, it is obvious that $P^{ki}|_{x \neq a}$ is the k-interleaved CSP of $P|_{x \neq a}$. □

From Properties 4 and 5, we can derive the following important corollary.

Corollary 2. *Let $P = (X, D, C)$ be a CSP and $P^{ki} = (X^{ki}, D^{ki}, C^{ki})$ be the k-interleaved CSP of P. Let Δ be a set of value refutations on variables of X. If $Q = (X, D^Q, C^Q)$ is the GAC+kWC-closure of $P|_\Delta$ and $R = (X^{ki}, D^R, C^{ki})$ is the GAC-closure of $P^{ki}|_\Delta$, then we have $D^Q = D^R[X]$.*

Corollary 2 is central to our approach. It allows us to achieve DkWC indirectly using GAC at any stage of a backtracking search with the k-interleaved CSP generated at the beginning of the search. So, it is important to note that the generation of the k-interleaved CSP is only performed once since it can be used during the whole search.

The complexity of enforcing DkWC during the search (not considering the cost of the generation of the k-interleaved CSP) is the complexity of enforcing GAC on the k-interleaved CSP. As the k-interleaved CSP only contains table constraints, the complexity analysis will use the optimal time complexity for a table constraint given in [MVHD14b] and presented in Section 4.4. Let P be a CSP with n variables, a maximum domain size d, e constraints, a maximum number t of tuples allowed by a constraint, and a maximum constraint arity r. The k-interleaved CSP of P is a CSP with $n' = n + e$ variables, a maximum domain size $d' = max(d, t)$, $e' = e + \binom{e}{k}$ constraints[1], an upper bound $t' = t^k$ on the maximum number of allowed tuples by a constraint, and a maximum constraint arity $r' = max(r + 1, k)$ Enforcing GAC on the k-interleaved CSP with optimal table constraint propagators has a complexity of $O(e' \cdot (r' \cdot t' + r' \cdot d')) = O((\binom{e}{k} + e) \cdot (r' \cdot t' + r' \cdot d'))$.

Necessity of Hybrid Constraints. It is important to note that the filtering procedure for DkWC presented in this chapter is stronger than the propagation that would be obtained by simply replacing the original constraints by their joins. The reason is that in the second setting, the invalidation of a tuple in a join is not reflected in the other joins, whereas with the k-interleaved CSP, the supports for the tuples on a join must themselves be supported. This is illustrated by Example 8.

1 $\binom{e}{k}$ is the binomial coefficient corresponding to the number of subsets of size k that can be formed using elements from a set of size e.

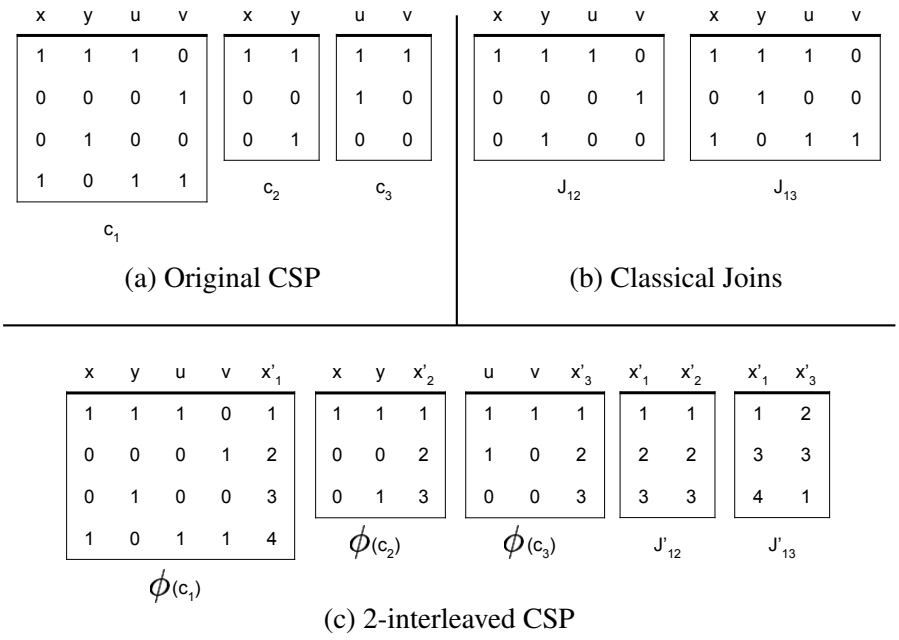


Figure 6.1: Illustration of (a) the CSP, (b) the classical joins and (c) the 2-interleaved CSP from Example 8

Example 8. Let $P = (X, D, C)$ be a CSP such that $X = \{x, y, u, v\}$, $D = \{0,1\}^4$ and $C = \{c_1, c_2, c_3\}$ with $scope(c_1) = \{x, y, u, v\}$, $scope(c_2) = \{x, y\}$ and $scope(c_3) = \{u, v\}$, and $table(c_1)$, $table(c_2)$ and $table(c_3)$ defined as in Figure 8(a).

Let us compare domain pairwise consistency (D2WC) with the joins of any two pairs of constraints: in this CSP, the two possible joins and the 2-interleaved CSP are depicted in Figure 6.1. On the one hand, enforcing GAC on the two join constraints J_{12} and J_{13} has no effect (observe that values 0 and 1 are present in each column of both tables). On the other hand, enforcing GAC on the 2-interleaved CSP reduces $D(y)$ to $\{1\}$ and $D(v)$ to $\{0\}$. The reduction of $D(y)$ comes from the tuple $(0,0)$ in $table(c_2)$ which is the only support for $y = 0$ on c_2. This tuple is only supported in J'_{12} by the second tuple of c_1: $(0,0,0,1)$. As $(0,0,0,1)$ has no support on J'_{13}, we can safely remove 0 from $D(y)$.

6.4 PRACTICAL USE OF THE DOMAIN k-WISE CONSISTENCY

Enforcing GAC on the k-interleaved CSP may be expensive. One cause is the number of constraints from the k-dual CSP that are added to the k-interleaved CSP: $\binom{e}{k}$ for an original CSP with e constraints. Some of those constraints can be ignored without degrading the pruning. For instance, this is trivially the case for k-dual constraints that are based on original constraints sharing no variables. Of course, a trade-off can be made between propagation strength and time complexity by integrating a subset of the possible $\binom{e}{k}$ constraints. In that case, the pruning achieved will be weaker than DkWC. Suppose that we limit the integration to the p most promising constraints from the k-dual CSP. The complexity, following our analysis performed above, becomes $O((e+p)\cdot(r'\cdot t'+r'\cdot d'))$: the term $\binom{e}{k}$ has been replaced by p. The most promising constraints can be selected, for example, according to the size of the joins. Indeed, small joins are more likely to prune more the search space whereas large joins are a cause of inefficiency, as it is related to t'.

Following this discussion, we propose two weak variants of DkWC, and refer to them as *weak* DkWC: they only consider a subset of all possible k-dual constraints. The first one, called DkWC^{cy} only considers constraints from the k-dual CSP corresponding to cycles in the original constraint graph (i.e., sequences of constraints at least sharing variables with previous and next constraints in a circular manner). There are typically far less cycles of k constraints than combinations of k constraints and besides they usually form smaller joins. The consistency level attained by DkWC^{cy} is weaker than DkWC but in practice, as we shall see, it shows good performances. Since all the original constraints are included in DkWC^{cy}, the consistency level attained is stronger than GAC. Unsurprisingly, for some problems, the size of the joins of some cycle constraints may be too large to be treated efficiently. For instance, in the modifiedRenault benchmark, some joins computed from cycles of length 3 exceed 10^6 tuples. This is why the second variant of DkWC, called DkWC^{cy-}, only considers constraints from the k-dual CSP corresponding to cycles in the original constraint graph and for which the size of the join constraint is smaller than a specified parameter (e.g., a percentage of the size of the largest table). In other words, a maximum size is imposed on the size of the joins and the joins exceeding that limit are not included in the k-interleaved CSP. The consistency level attained by DkWC^{cy-} is weaker than DkWC and DkWC^{cy}, but its practical interest will be shown on some problems. DkWC^{cy-} attains a level of consistency stronger than GAC.

6.5 EXPERIMENTAL RESULTS

This section presents some experimental results concerning DkWC. For each test, we propose to maintain DkWC on the CSP (i.e., GAC on the k-interleaved CSP) at each node of the search trees developed by a backtracking search. However, as discussed in Section 6.4, including all k-dual constraints is unpractical for many problems, because of the number of additional constraints and/or because of their size. In our experiments, we thus only use the weaker versions of the filtering, namely, DkWC^{cy} and DkWC^{cy-}, and we have focused our attention on weak D3WC and weak D4WC. Those values of k allow a significant search space reduction with respect to GAC while keeping the number and size of k-dual constraints tractable. Notice that labeling is only performed for the original variables during a search, and that all solutions are searched for. The GAC propagator used for the original constraints as well as the k-dual ones is the optimal state-of-the-art propagator from [MVHD14b], presented in Section 4.4.

Our filtering procedure is compared with the GAC propagator AC5TCOpt-Sparse from Section 4.4, the maxRPWC3 procedure from [BSW08], and the state-of-the-art eSTRw propagator from [LPS13]. The eSTRw propagator is weaker than eSTR but easier to incorporate into an existing solver, and is at least as good as eSTR on the benchmarks used in [LPS13]. All the algorithms are (re-)implemented on top of Comet but it is unfortunately impossible to implement the filtering algorithm from [KWR$^+$10] in this solver. The reason is that the context management of Comet prohibits the start of an independent backtracking search inside a propagator, as [KWR$^+$10] requires. Eight different benchmarks have been used. Two of them contain only binary table constraints, five of them contain binary and ternary table constraints, and the last benchmark contains table constraints up to arity 10. The tests were executed on an Intel Xeon 2.53GHz using Comet 2.1.1. A timeout of 20 minutes on the total execution time was used for each instance. When comparing different techniques in terms of CPU time and search space sizes, we can only use the subset of instances for which none of the techniques timed out. In the results, we thus do not report measurements for some of the techniques on some benchmarks because including them would cause the common instance set to be empty or too small for a meaningful comparison. In the tables, a '-' thus represents a technique that timed out on the set of instances considered. The global percentage of the instance set that is solved is however given for each technique on each benchmark.

The results are presented in Table 6.1. For each instance set and each technique, we present the means of different quantities (times are in seconds): the

execution time (T), the "posting" time (pT), the join selection time (jST) that corresponds to the amount of time used to select the joins for the k-interleaved CSPs, the join computation time (jT), the number of propagator calls (nP), the number of fails (nF), and the number of choice points (nC). Table 6.1 also presents the percentage to the best with respect to execution time (%b), the mean of the percentage to the best instance by instance (μ%b), and the percentage of instances from the sets that are solved (%sol). The total time (T) includes all precomputations the algorithms have to perform before the search. This means that both the times of join selection (jST) and the join computation (jT) for our DkWC algorithm are included in T. The posting time (pT) is the time taken between the loading of the instance file and the start of search without the time for jT. It thus includes the time for all the precomputations except for the join computation. The difference between %b and μ%b is the following. For %b, all execution times are averaged before computing it: there is thus one identified best algorithm. For μ%b, the percentages are first computed instance by instance, and then aggregated with a geometric mean (as suggested in [FW86]): this measure takes into account the fact that different instances may have different best algorithms.

Binary Random Instances This instance set contains 50 instances involving binary table constraints. These instances have 50 variables, a uniform domain size of 10, and 166 constraints, whose proportion of allowed tuples is 0.5. They have been generated using the model RD [XBHL07], in or close to the phase transition. The search strategy used to solve them (for all techniques) was a lexicographic variable and value ordering. A plot of the percent best quantities can be found in Figure 6.2. In this figure, wDkWC represents D3WC^{cy}. On this benchmark, D3WC^{cy} includes on average 48.4 3-dual constraints, and their tables contain on average 111.5 tuples. We can see that the pruning obtained by D3WC^{cy} on this benchmark allows it to drastically reduce the search space. Moreover, since the mean number of added constraints from the 3-dual CSP and their size is small, D3WC^{cy} has the lowest overall computation time. Propagators maxRPWC3 and eSTRw also reduce the search space with respect to GAC (partly due to the presence of constraints with identical scopes), but this reduction comes at the price of a greater total computation time.

Ternary Random Instances This instance set contains 50 instances involving ternary table constraints. These instances have 50 variables, a uniform domain size of 5, and 75 constraints, whose proportion of allowed tuples is 0.66. They have been generated using the model RD [XBHL07], in or close to the phase transition. The search strategy used to solve them was a lexicographic

propagator	T	pT	jST	jT	nP	nF	nC	%b	μ%b	%sol
Binary Random										
GAC	9.9	0.0	0.0	0.0	3 M	7.7 k	1 213.2	337	218	**100**
maxRPWC3	72	0.7	0.0	0.0	148 k	2.2 k	340.1	2448	1833	98
eSTRw	11.4	0.1	0.0	0.0	293 k	2.2 k	340.1	389	283	**100**
D3WC^{cy}	**2.9**	0.1	0.0	0.1	963 k	0.5 k	68.4	100	113	**100**
Ternary Random										
GAC	23.1	0.0	0.0	0.0	4 M	42.4 k	11.8 k	183	223	**100**
maxRPWC3	124	0.2	0.0	0.0	237 k	8.2 k	2.2 k	982	1455	90
eSTRw	16.6	0.0	0.0	0.0	409 k	7.7 k	2.1 k	131	189	**100**
D3WC^{cy}	**12.6**	0.5	0.1	0.4	2 M	0.6 k	0.1 k	100	143	**100**
AIM										
GAC	82	0.1	0.0	0.0	35 M	941.6 k	522 k	6745	460	46
maxRPWC3	14.7	1.0	0.0	0.0	46 k	1.2 k	0.7 k	1204	1481	46
eSTRw	**1.2**	0.3	0.0	0.0	35 k	0.7 k	0.4 k	100	208	50
D3WC^{cy}	3.4	2.4	1.2	0.5	139 k	0.1 k	0.1 k	279	497	**88**
Pret										
GAC	160	0.0	0.0	0.0	58 M	7 M	5 M	121	121	**50**
maxRPWC3	977	0.0	0.0	0.0	26 M	7 M	5 M	741	741	**50**
eSTRw	504	0.0	0.0	0.0	30 M	7 M	5 M	382	382	**50**
D3WC^{cy}	**132**	0.0	0.0	0.0	57 M	4 M	3 M	100	100	**50**
Langford-2										
GAC	**0.5**	0.0	0.0	0.0	171 k	1.4 k	1 k	100	100	**58**
maxRPWC3	44.6	2.2	0.0	0.0	43 k	1.4 k	1 k	9637	3863	46
eSTRw	1.5	0.1	0.0	0.0	65 k	1.4 k	1 k	326	233	54
D3WC^{cy}	10.5	0.9	0.1	3.2	2 M	0.7 k	0.7 k	2270	1782	50
Dubois										
GAC	793	0.0	0.0	0.0	158 M	42 M	37 M	394	390	15
maxRPWC3	-	-	-	-	-	-	-	-	-	8
eSTRw	598	0.0	0.0	0.0	37 M	5 M	3 M	297	294	15
D4WC^{cy}	**201**	0.1	0.0	0.0	68 M	2 M	1 M	100	100	**30**
TSP-20										
GAC	**52**	0.5	0.0	0.0	14 M	17 k	7 k	100	100	**93**
maxRPWC3	-	-	-	-	-	-	-	-	-	33
eSTRw	233	1.5	0.1	0.0	5 M	17 k	7 k	447	438	80
D3WC^{cy}	-	-	-	-	-	-	-	-	-	40
D3WC^{cy-}	94	4.3	0.6	0.1	40 M	17 k	7 k	180	270	**93**
Modified Renault										
GAC	-	-	-	-	-	-	-	-	-	6
eSTRw	-	-	-	-	-	-	-	-	-	0
maxRPWC3	743	0.0	0.0	0.0	23.7	0.0	0.0	148	417	26
D3WC^{cy}	-	-	-	-	-	-	-	-	-	0
D3WC^{cy-}	**502**	497	3.9	5.1	33 k	0.0	0.0	100	111	**34**

Table 6.1: Experimental results of our weak DkWC propagators.

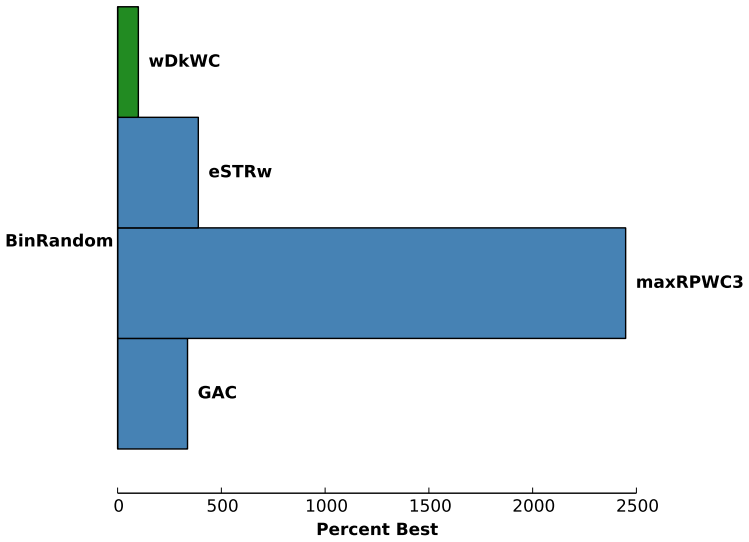


Figure 6.2: Percent best quantities for the binary random instance set

variable and value ordering. A plot of the percent best quantities can be found in Figure 6.3. In this figure, wDkWC represents D3WC^{cy}. On this benchmark, D3WC^{cy} includes, on average 112.3 3-dual constraints, and their tables contain on average 529.5 tuples. As for the binary case, the search space reduction obtained by D3WC^{cy} is important. On this benchmark, the size and number of added constraints is small enough to allow D3WC^{cy} to be the fastest technique. Note that the search space reduction obtained by maxRPWC3 doesn't repay its cost, in contrast to eSTRw.

AIM Instances This instance set contains 24 instances from the AIM series used in the CSP solver competition [vDLR] (100 variables, a majority of ternary constraints and binary ones). The search strategy used was a lexicographic variable and value ordering. D3WC^{cy} includes 3000 constraints from the 3-dual, on average, and the added constraints contain on average 30.3 tuples. Plots of the percent best quantities can be found in Figure 6.4. In this figure, wDkWC represents D3WC^{cy}. On this benchmark, the filtering obtained by maxRPWC3, eSTRW and D3WC^{cy} allows each of them to be significantly faster than GAC. Although D3WC^{cy} achieves the best search space reduction, eSTRw remains the fastest technique. The greater computation time for D3WC^{cy} is due to the number of 3-dual constraints included in the

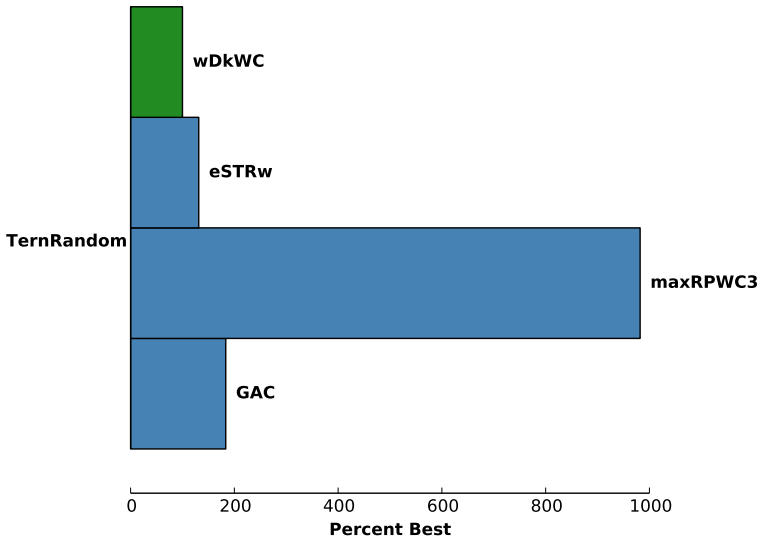


Figure 6.3: Percent best quantities for the ternary random instance set

3-interleaved CSPs: 3000 on average while the number of original constraints lies between 150 and 570. Even if the 3-dual constraints have small tables, they still have to be propagated during the search. Interestingly, D3WC^{cy} solves significantly more instances than the other techniques.

Pret Instances This instance set also comes from the CSP solver competition [vDLR] and has 8 instances (only ternary table constraints). The search strategy used was a lexicographic variable and value ordering. D3WC^{cy} includes on average 13 constraints, that have a mean size of 8 tuples. Figure 6.5 plots the percent best quantities for this benchmark. In this figure, wDkWC represents D3WC^{cy}. On this benchmark, neither maxRPWC3 nor eSTRw is able to reduce the search space with respect to GAC. Their additional computations make them slower than GAC. The small number of small constraints from the 3-dual CSP included by D3WC^{cy} allows it to significantly reduce the search space and to be the fastest on this series. The mean percentage to the best ($\mu\%b$) of D3WC^{cy} means that it is the best technique on average but also on each instance. Note that the join selection, join computation and posting times are negligible for this problem.

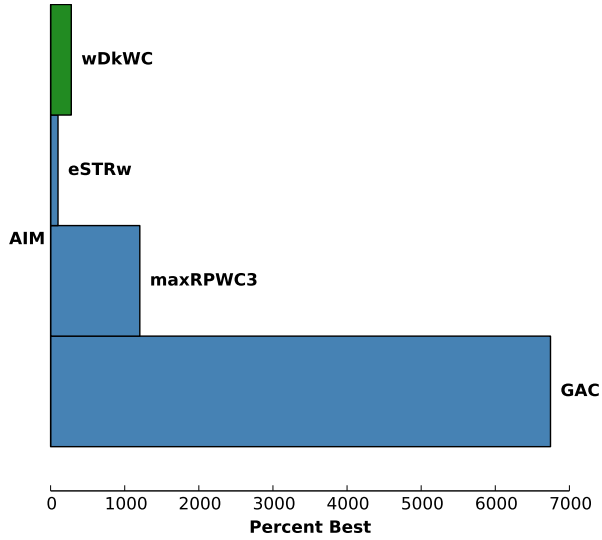


Figure 6.4: Percent best quantities for the aim instance set

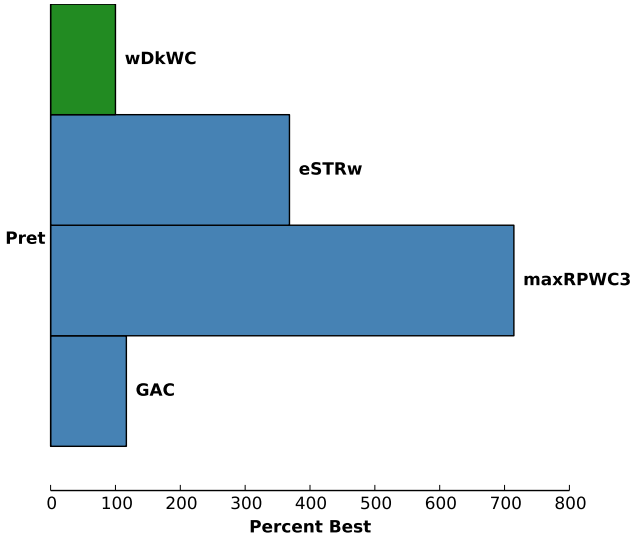


Figure 6.5: Percent best quantities for the pret instance set

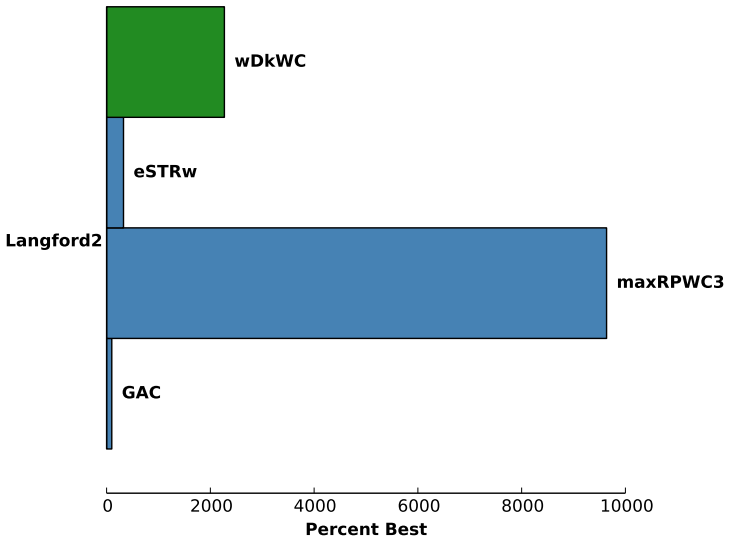


Figure 6.6: Percent best quantities for the Langford-2 instance set

Langford Number Problem The Langford number problem is Problem 24 of CSPLIB[2], here modeled with binary table constraints only. We used the set Langford-2 containing 24 instances that can be found in [Lec]. The search strategy used was *dom/deg* combined with a lexicographic value ordering. On this set, D3WC^{cy} includes, on average, 328 3-dual constraints, whose tables contain 274.7 tuples on average. Figure 6.6 plots the percent best quantities for this benchmark. In this figure, wDkWC represents D3WC^{cy}. On this benchmark, GAC is the fastest propagator on each instance. Neither maxRPWC3 nor eSTRw is able to reduce the search space with respect to GAC. However, they have a lower propagator call count. This is due to their ability to reach the fixed point faster. The number of constraints added from the 3-dual by D3WC^{cy} is large compared to the number of original constraints (the non-timeout instances are the smallest ones). The small search space reduction obtained by D3WC^{cy} does not compensate for the cost of propagating all the added constraints. The number of propagator calls is significantly larger for D3WC^{cy}. We can also see that, on this benchmark, the time required to compute the joins is larger than the time required by GAC to solve the instances.

2 www.csplib.org

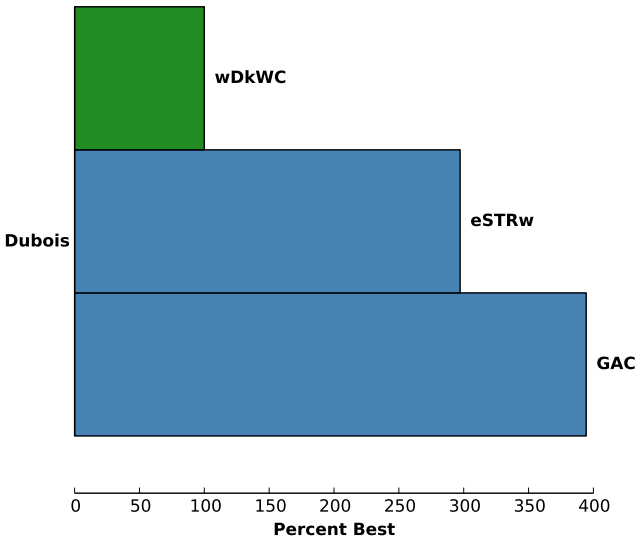


Figure 6.7: Percent best quantities for the Dubois instance set

Dubois Instances Those 13 instances also comes from the CSP solver competition [vDLR] (ternary table constraints). These instances do not contain any cycle of original constraints of length 3. We thus present the results of D4WC^{cy}. A plot of the percent best quantities can be found in Figure 6.7. In this plot, wDkWC represents D4WC^{cy}. On this series, D4WC^{cy} adds on average 156 4-dual constraints and their tables contain, on average, 15.2 tuples. Clearly, D4WC^{cy} is the fastest approach here and solves more instances than the other techniques. D4WC^{cy} is also the fastest on each instance, as shown by the mean percentage to the best (μ%b). The search space reduction obtained by eSTRw is less than that obtained by D4WC^{cy} but it allows it to be faster than GAC.

Travelling Salesman Problem We used the set of 15 Travelling Salesman satisfaction instances *tsp-20* from [Lec] (table constraints of arity 2 and 3). The search strategy used here was *dom/deg* combined with a lexicographic value ordering. On this instance set, there are, on average, 1000 cycles of length 3 in the 3-dual CSP and they contain up to 2000 tuples. In that context, D3WC^{cy} only solves 40% of the instances. We thus present the results for D3WC^{cy-} where the limit on the size of the joins is set to one percent of the maximal original constraint size (200). Figure 6.8 plots the percent best quantities for this benchmark. In this figure, wDkWC represents D3WC^{cy-}.

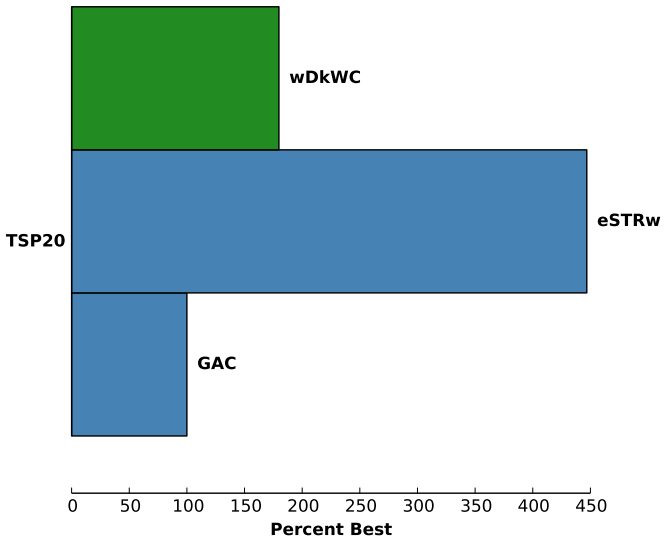


Figure 6.8: Percent best quantities for the travelling salesman instance set

D3WC^{cy-} includes 59.8 constraints from the 3-dual CSP on average, and their tables contain, on average, 26 tuples. As we can see, on those instances, neither eSTRw nor D3WC^{cy-} is able to reduce the search space. The extra computations of eSTRw and the extra propagation effort of D3WC^{cy-} make them slower than GAC (which is also the fastest approach on each instance). However, D3WC^{cy-} is faster than eSTRw and it is the only one able to solve the same number of instances as GAC.

Modified Renault Problem The modified Renault problem instances originate from a real Renault Megane configuration problem, modified to generate 50 instances [Lec] (large tables and arities up to 10). The search strategy used was *dom/deg* variable ordering combined with a lexicographic value ordering. Since the tables of the original problem can have up to 50K tuples, D3WC^{cy} is unpractical because of the size of the joins. We thus present the results for D3WC^{cy-} where the limit on the size of the joins has been set to one percent of the largest original constraint size (500), as in the TSP benchmark. Plots of the percent best quantities can be found in Figure 6.9. In this figure, wDkWC represents D3WC^{cy-}. On those instances, D3WC^{cy-} includes 481.6 3-dual constraints on average, and their tables contain on average 253.9 tuples. As we can see, both maxRPWC3 and D3WC^{cy-} detect the inconsistencies of all

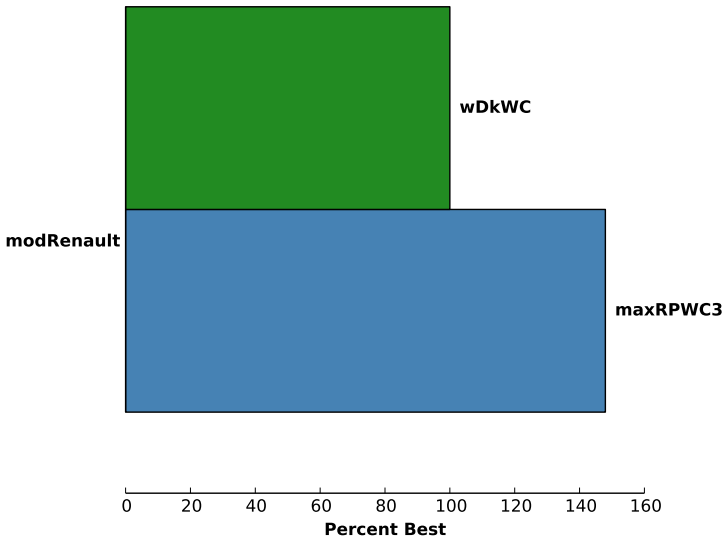


Figure 6.9: Percent best quantities for the modified Renault instance set

instances without performing any search ($nC = nF = 0$). However, despite the fact that D3WC^{cy-} has a larger propagator call count, it is faster than maxRPWC3. D3WC^{cy-} is also able to solve more instances than maxRPWC3.

Summary of the Experimental Results A summary of the experimental results can be found in Table 6.2. This table contains the total execution time (T) and the percentage of instances solved (% sol) for each technique. The column wDkWC represents our weak DkWC approach: it is D3WC^{cy} for binary random, ternary random, AIM, Pret and Langford-2 instances, D4WC^{cy} for Dubois instances and D3WC^{cy-} for TSP-20 and modified Renault instances. Weak DkWC is faster than maxRPWC3 and eSTRw, except for two benchmarks. It is also faster than GAC on all but two benchmarks, where GAC is faster than all strong consistencies. Weak DkWC is also the strong consistency leading to the largest reductions of search space. Except on Langford-2, weak DkWC solves the largest number of instances within the time limit.

For all these benchmarks, we insist that (full) DkWC cannot be used in practice because of the number of possible joins and/or their size. This is the reason why we have introduced weak DkWC. On the majority of benchmarks, we used DkWC^{cy} but on two benchmarks, even D3WC^{cy} suffers from the number of 3-dual constraints and their sizes. Consequently, we also used DkWC^{cy-},

Benchmark	GAC		maxRPWC3		eSTRw		wDkWC	
	T	%sol	T	%sol	T	%sol	T	%sol
Binary Random	9.9	**100**	72	98	11.4	**100**	**2.9**	**100**
Ternary Random	23.1	**100**	124	90	16.6	**100**	**12.6**	**100**
AIM	82	46	14.7	46	**1.2**	50	3.4	**88**
Pret	160	**50**	977	**50**	504	**50**	**132**	**50**
Langford2	**0.5**	**58**	44.6	46	1.5	54	10.5	50
Dubois	793	15	-	8	598	15	**201**	**30**
TSP-20	**52**	**93**	-	33	233	80	94	**93**
ModRenault	-	6	743	0	-	26	**502**	**34**

Table 6.2: Summary of the results of the experimental section. T is the total solving time in seconds and %sol is the percentage of the instances solved

for which the best limit on the joins size has been empirically found to be equal to 1 percent of the maximum original constraint size. This parameter value allows D3WC^{cy-} to include a significant number of small (highly filtering) 3-dual constraints without including too many of them. All these results show, on a large variety of benchmarks with constraints of various arities, that the weak DkWC filtering procedures defined in this chapter are competitive.

6.6 STATISTICAL TREATMENT OF THE EXPERIMENTS

This section is concerned with the application of the statistical procedure presented in Chapter 3 to the experimental results for the propagators in this chapter. In this context, as timeouts are present and there are several classes of instances, we will use the θ_5 statistic of interest. Recall that θ_5 is the mean width of the area between the multi-class cumulative distributions of the algorithms being compared. This represents the mean, over the different proportions of the whole instance set, of the differences in the times the algorithms take to solve given proportions of the set. In the multi-class empirical distributions, each class of instances has a weight. The weights used in this section are designed to give less importance to the fully random datasets and to give more importance to the non-academic ones (modified Renault and TSP). The same rule has been used in Chapter 4 to determine the weights of the datasets. Again, larger classes of instances do not weigh more in the value of $\hat{\theta}_5$ since the number of

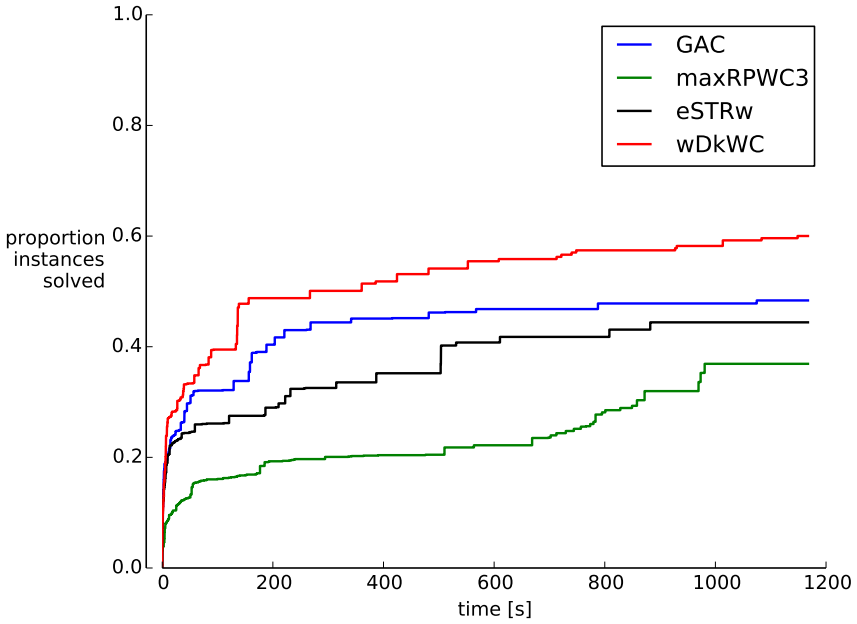


Figure 6.10: Non-bootstrapped multi-class empirical cumulative distributions of the weak DkWC filtering procedure versus the state of the art

instances of each class is inluded in the denominator of the class contribution to $\hat{\theta}_5$. To illustrate the results, Figure 6.10 gives the non-bootstrapped multi-class empirical cumulative distribution of our weak DkWC filtering procedure versus the state of the art. Recall that the multi-class empirical cumulative distribution of algorithm A gives, for each time t, the weighted proportion of the full instance set that has been solved by A in a time less than or equal to t. As we can see in Figure 6.10, our weak DkWC filtering procedure is the fastest and is also the technique resulting in the largest number of instances solved at the end. Surprisingly, GAC is the second fastest filtering technique on the full instance set.

Bootstrapping was used on the estimator of θ_5 to get a confidence interval. The results of the pairwise comparisons of the algorithms are given in Table 6.3. The confidence intervals are written with the lower bound on top of the upper bound and significant positive confidence intervals are written in bold. For those confidence intervals, 10,000 bootstrap samples were drawn from the sets of all experimental results and a confidence level α of 0.05 was used. The numbers correspond to seconds. The entry in the table in the line of algorithm

	GAC	maxRPWC3	eSTRw	wDkWC
GAC	–	**286.5**	**77.1**	-110.5
		473.8	**194.4**	-32.3
maxRPWC3	-473.8	–	-395.1	-498.6
	-286.5		-237.4	-302.4
eSTRw	-194.4	**237.4**	–	-224.4
	-77.1	**395.1**		-94.7
wDkWC	**32.3**	**302.4**	**94.7**	–
	110.5	**498.6**	**224.4**	

Table 6.3: Confidence intervals for the θ_5 statistic of interest on the propagators from this Chapter. The values correspond to seconds and the confidence intervals are given with the lower bound on top of the upper bound.

A and the column of algorithm B gives the confidence interval for $\theta_5(A, B)$. Positive values correspond to A's being faster than B in general (all the instances included). If 0 is outside the confidence interval for $\theta_5(A, B)$, A and B perform significantly differently. As in table 6.2, wDkWC stands for our weak DkWC approach.

In Table 6.3, we can see that when aggregating the results of all the datasets, our weak DkWC propagator is significantly faster than all the other propagators. The GAC propagator is the second fastest propagator: it is significantly faster than all the propagators, except our weak DkWC one. MaxRPWC3 is significantly slower than all the other propagators. The big difference between Table 6.3 and Table 6.2 is that Table 6.3 includes the information of all the instances, including instances for which one or more algorithm timeouts. In Table 6.2, the algorithms are only compared on the set of instances that are solved by all of them. The information on the other instances is only reflected through the %sol quantity. In Table 6.3, those two pieces of information are combined: if an instance is in a bootstrap set and solved by only one of the two compared algorithm, it will advantage the algorithm solving it since it will be included in its MECD before the timeout.

CONCLUSION AND PERSPECTIVES

CONCLUSION

This thesis presented different chapters, each of which has its own conclusion.

Efficient and Optimal GAC Propagators for Table Constraints

This chapter proposed five different value-based GAC propagators for table constraints, using the AC5 generic framework. Two of them (AC5TCOtp-Tr and AC5TCOpt-Sparse) have an optimal time complexity. Those propagators record, for every value of the variables, the index of its first current support in the table. They also use, for each variable of a tuple, the index of the next tuple sharing the same value for this variable. They differ in their use of the information on the validity of the tuples as well as the order of the tuples in the formed next chains. AC5TCOpt-Sparse is the best of our value-based algorithms. As does AC5TCOpt-Tr, AC5TCOpt-Sparse embeds the Q-validity information into the indexing structure, avoiding unnecessary visits of invalid tuples and leading to an optimal algorithm with a time complexity of $O(r \cdot t + r \cdot d)$ per table constraint. However, AC5TCOpt-Sparse relaxes the requirement that the tuples in the structure are ordered as they are in the table. Doing that allows AC5TCOpt-Sparse to have far less backtrackable structures. While not changing its theoretical complexity, this relaxation allows AC5TCOpt-Sparse to be more efficient in practice. Our other algorithms have a time complexity of $O(r^2 \cdot t + r \cdot d)$ per table constraint. The experimental results show that on instances containing only binary table constraints, our algorithms are outperformed by AC3rm. AC3rm is designed only for binary constraints. However, they are faster than the other state of the art table constraint propagators STR2+, STR3 and MDD^c. As the arity of the tables in the instances goes up to 4, our propagators are the fastest ones. However, when the arity of the table increases further, the conclusion changes. The existing state-of-the-art propagators STR2+, STR3 and MDD^c are faster than our algorithms on one different benchmark each. When all the benchmarks are used together to assess the performances of the algorithms with the statistical procedure defined in Chapter 3, our optimal AC5TCOpt-Sparse is significantly faster than all the other propagators.

The Smart Table Constraint

This chapter presented a new constraint, the Smart Table Constraint, generalizing existing table constraints by replacing tuples with smart tuples. Smart

tuples are allowed to contain simple arithmetic constraints, making the representation of constraints compact and natural. A variable in a smart tuple can thus take many values, reducing the length of the table, as with the use of compressed tuples and short supports for table constraints. Furthermore, the constraints in the smart tuples encode the relations that exist between columns in the table, further reducing the length of the smart tables. Each smart tuple is a small CSP and the literals that it supports are the solution to its small CSP. Hence, for the sake of keeping the propagation tractable, a restriction on the form of the smart tuples is imposed: the smart tuples have to form binary acyclic networks. A GAC propagator, based on STR/STR2, has been defined for smart table constraints. It is called SmartSTR2: it efficiently processes the smart tuples, using the restriction on their form to do so. The experimental results on the encoding into smart table constraints of global constraints versus their dedicated propagators show the interest of SmartSTR2.

Efficient Filtering Procedure for Domain k-Wise Consistency on Table Constraints

In this chapter, we combined two consistencies: generalized arc consistency and k-wise consistency. Generalized arc consistency filters the domains of the variables, while k-wise consistency filters directly the constraints. The integration of a constraint filtering consistency in an existing solver can be complicated. We have thus derived a domain-filtering consistency, the Domain k-wise Consistency, from the combination of kWC and GAC. This consistency is stronger than GAC. More importantly for such a strong consistency, we have shown how to establish and maintain it by establishing and maintaining GAC on a modified CSP, called the k-interleaved CSPs. Such reformulated CSPs incorporate dual variables representing the original constraints, hybrid constraints linking the original constraints and dual variables, and k-dual constraints constraining the dual variables. The k-interleaved CSPs are simple to generate, and need to be generated only once before the search. Enforcing GAC on the k-interleaved CSP enforces domain k-wise consistency on the original CSP. The k-dual constraints represent the joins of the original constraints. To manage the complexity and the number of join operations, we have proposed a few solutions, such as the ones relying on the presence of cycles in the original constraint graph or on the use of a limit on the maximal size of joins. The experimental results that we have obtained show, for a large variety of problems, that our weak DkWC filtering procedures are competitive. When aggregating all the experimental data with the statistical procedure from Chapter 3, we can

see that our weak DkWC filtering procedure is significantly faster than the other approaches.

PERSPECTIVES

There exist many different perspectives opened up by the research presented in this thesis. Some of them are listed below:

- **Development of non-optimal but efficient GAC propagators for table constraints**: the performances of the AC5TC-Recomp, STR2+ and MDD^c GAC propagators for table constraints are competitive on many benchmarks. This suggests that there is room to develop propagators for table constraints that are non-optimal but efficient. Indeed, the optimality of many propagators is based on large data structures that have to be backtracked. Without the optimality requirement, the need for data structures could be reduced, leading to more efficient propagators. A starting point could be to inspect the effects of not backtracking some structures in the optimal propagators. The information would have to be recomputed.

- **Propagators using the *collect and propagate* paradigm**: value-based propagators have information on the literal being removed at call time. However, this requires them to be called for each removed literal. In contrast, constraint-based propagators only have the information that some literal(s) have been removed. Having less information allows those propagators to be called far less often than value-based ones. The *collect and propagate* paradigm lies between the constraint and the value-based ones: propagators have information about all the literals that have been removed since their last call. This paradigm could lead to propagators that combine the efficiency of the value-based ones (information on the literals removed) with the advantage of the constraint based ones (one call summarizes many literal removals).

- **Combination of smart tables**: a smart table constraint can be seen as a disjunction of smart tuples. They allow efficiently encoding many well known global constraints. Combining two smart tables into one table constraint would encode the conjunction of the two constraints, since the shared variables would have to agree. Combining two smart table constraints corresponding to global constraints would thus represent the conjunction of the global constraints into one constraint. Combined

smart table constraints would hence improve the pruning of the search tree.

- **Automatic translation of table constraints into smart table constraints**: The smart table constraints presented in this thesis are directly encoded by the user. As a smart table constraint can encode any constraint efficiently, it would be interesting to investigate the possibility of automatically translating classical table constraints into smart ones. Smart table constraints could reduce the space and time consumption of table constraints. However, this task would require detecting the relations between the columns in classical table constraints, which is not an easy task.

- **Practicality of domain k-wise consistency**: as seen in Chapter 6, the full DkWC is too costly for most practical applications. This is even more true as k increases. As a consequence, two weak variants of our filtering have been proposed, relying on the selection of interesting constraints and a limit on their size. However, the choice of constraints, as well as the parameter k, has a tremendous effect on the effectiveness of the technique. A very interesting perspective for DkWC would be to investigate the best set of constraints to include in the k-interleaved CSP, and which k to use for which problem.

- **Strong consistency for hard problems**: the practical use of strong consistency levels is more and more studied. The problem with strong consistency is that it is expensive to compute and only pays off when the search space reduction is large enough to cover the cost of the propagation. Perspectives for this concern are multiple. One could concentrate on the development of efficient procedures, achieving a (maybe uncharacterized) relaxed version of a strong consistency level. This is the case for our weak DkWC propagators. Another possible perspective for the strong consistency levels would be the characterization of the problems/instances where a given strong consistency level is beneficial. This would allow selecting the appropriate level of consistency based on the characteristics of the problem or the instance. A last perspective would be to use different levels of propagation at different stages of the search in order to only pay the cost of strong propagation where it has more chances of greatly pruning the search space.

BIBLIOGRAPHY

[BCDP05] Nicolas Beldiceanu, Mats Carlsson, Romuald Debruyne, and Thierry Petit. Reformulation of global constraints based on constraints checkers. *Constraints*, 10(4):339–362, 2005.

[Bee06] Peter van Beek. Backtracking search algorithm. In Rossi et al. [RBW06].

[Bes94] Christian Bessière. Arc-consistency and arc-consistency again. *Artificial intelligence*, 65(1):179–190, 1994.

[Bes06] Christian Bessière. Constraint propagation. In Rossi et al. [RBW06].

[BFR99] Christian Bessière, Eugene C Freuder, and Jean-Charles Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107(1):125–148, 1999.

[BR97] Christian Bessière and Jean-Charles Régin. Arc consistency for general constraint networks: Preliminary results. In *IJCAI (1)*, pages 398–404, 1997.

[BR98] Christian Bessière and Jean-Charles Régin. Local consistency on conjunctions of constraints. In *Proceedings of ECAI'98 Workshop on Non-binary constraints*, pages 53–59, 1998.

[BR99] C. Bessière and J.-C. Régin. Enforcing arc consistency on global constraints by solving subproblems on the fly. In *Proceedings of CP'99*, pages 103–117, 1999.

[BR01] Christian Bessière and Jean-Charles Régin. Refining the basic constraint propagation algorithm. In *IJCAI*, volume 1, pages 309–315, 2001.

[BRYZ05] Christian Bessière, Jean-Charles Régin, Roland HC Yap, and Yuanlin Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.

[BSW08] C. Bessière, K. Stergiou, and T. Walsh. Domain filtering consistencies for non-binary constraints. *Artificial Intelligence*, 72(6-7):800–822, 2008.

[BT93] P. Briggs and L. Torczon. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 2(1-4):59–69, 1993.

[BW05] Fahiem Bacchus and Toby Walsh. Propagating logical combinations of constraints. In *IJCAI*, pages 35–40, 2005.

[Car06] Mats Carlsson. Filtering for the case constraint. Talk given at the advanced school on global constraints, 2006.

[CC95] Björn Carlson and Mats Carlsson. Compiling and executing disjunctions of finite domain constraints. In *Proceedings of ICLP'95*, pages 117–131, 1995.

[CJ98] Assef Chmeiss and Philippe Jégou. Efficient path-consistency propagation. *International Journal on Artificial Intelligence Tools*, 7(02):121–142, 1998.

[CY10] Kenil Cheng and Roland Yap. An mdd-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15:265–304, 2010.

[Dav97] Anthony Christopher Davison. *Bootstrap methods and their application*, volume 1. Cambridge university press, 1997.

[DB01] R. Debruyne and C. Bessière. Domain filtering consistencies. *Journal of Artificial Intelligence Research*, 14:205–230, 2001.

[DM02] Elizabeth D Dolan and Jorge J Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2):201–213, 2002.

[dSMDS11] Vianney le Clément de Saint-Marcq, Yves Deville, and Christine Solnon. An efficient light solver for querying the semantic web. In *Principles and Practice of Constraint Programming–CP 2011*, pages 145–159. Springer, 2011.

[dSMDSC12] Vianney le Clément de Saint-Marcq, Yves Deville, Christine Solnon, and Pierre-Antoine Champin. Castor: a constraint-based sparql engine with active filter processing. In *The Semantic Web: Research and Applications*, pages 391–405. Springer, 2012.

[DV10] Yves Deville and Pascal Van Hentenryck. Domain consistency with forbidden values. In *Proceedings of CP 2010*, pages 191–205. Springer, 2010.

[DVHM13] Yves Deville, Pascal Van Hentenryck, and Jean-Baptiste Mairy. Domain consistency with forbidden values. *Constraints*, 18(3):377–403, 2013.

[Efr79] Bradley Efron. Bootstrap methods: another look at the jackknife. *The annals of Statistics*, pages 1–26, 1979.

[ET94] Bradley Efron and Robert J Tibshirani. *An introduction to the bootstrap*, volume 57. CRC press, 1994.

[FHK+02] A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Global constraints for lexicographic orderings. In *Proceedings of CP'02*, pages 93–108, 2002.

[FHK+06] A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Propagation algorithms for lexicographic ordering constraints. *Artificial Intelligence*, 170(10):803–834, 2006.

[FW86] Philip J. Fleming and John J. Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, March 1986.

[GHLR14] Nebras Gharbi, Fred Hemery, Christophe Lecoutre, and Olivier Roussel. Sliced table constraints: combining compression and tabular reduction. In *Proceedings of CPAIOR'14*, pages 120–135, 2014.

[GJM06] Ian P. Gent, Chris Jefferson, and Ian Miguel. Watched literals for constraint propagation in minion. In *Proceedings of CP 2006*, pages 182–197. Springer-Verlag, 2006.

[GJMN07] Ian P. Gent, Chris Jefferson, Ian Miguel, and Peter Nightingale. Data structures for generalised arc consistency for extensional constraints. In *Proceedings of the AAAI 07*, pages 191–197. AAAI Press, 2007.

[GSS11] Graeme Gange, Peter J. Stuckey, and Radoslaw Szymanek. Mdd propagators with explanation. *Constraints*, 16(4):407–429, 2011.

[Jég91] P. Jégou. *Contribution à l'étude des Problèmes de Satisfaction de Contraintes: Algorithmes de propagation et de résolution. Propagation de contraintes dans les réseaux dynamique*. PhD thesis, Université de Montpellier II, 1991.

[JJNV89] P. Janssen, P. Jégou, B. Nouguier, and M.-C. Vilarem. A filtering process for general constraint-satisfaction problems: achieving pairwise-consistency using an associated binary representation. In *Proceedings of IEEE Workshop on Tools for Artificial Intelligence*, pages 420–427, 1989.

[JMNP10] Christopher Jefferson, Neil CA Moore, Peter Nightingale, and Karen E Petrie. Implementing logical connectives in constraint programming. *Artificial Intelligence*, 174(16):1407–1429, 2010.

[JN13] Christopher Jefferson and Peter Nightingale. Extending simple tabular reduction with short supports. In *Proceedings of IJCAI'13*, pages 573–579, 2013.

[KB01] George Katsirelos and Fahiem Bacchus. GAC on conjunctions of constraints. In *Proceedings of CP'01*, pages 610–614, 2001.

[KW07] George Katsirelos and Toby Walsh. A compression algorithm for large arity extensional constraints. In *Proceedings of CP 2007*, pages 379–393. Springer-Verlag, 2007.

[KWR+10] S. Karakashian, R. Woodward, C. Reeson, B. Choueiry, and C. Bessière. A first practical algorithm for high levels of relational consistency. In *Proceedings of AAAI'10*, pages 101–107, 2010.

[Lec] Ch. Lecoutre. Instances of the constraint solver competition. http://www.cril.fr/∼lecoutre/.

[Lec09] Christophe Lecoutre. *Constraint Networks: Techniques and Algorithms*. ISTE/Wiley, 2009.

[Lec11] Christophe Lecoutre. Str2: optimized simple tabular reduction for table constraints. *Constraints*, 16:341–371, 2011.

[LH+07] Christophe Lecoutre, Fred Hemery, et al. A study of residual supports in arc consistency. In *Proceedings of IJCAI 2007*, volume 7, pages 125–130, 2007.

[Lho04] Olivier Lhomme. Arc-consistency filtering algorithms for logical combinations of constraints. In *Proceedings of CPAIOR 2004*, pages 209–224, 2004.

[Lho12] O. Lhomme. Practical reformulations with table constraints. In *Proceedings of ECAI'12*, pages 911–912, 2012.

[LLY12] C. Lecoutre, C. Likitvivatanavong, and R.H.C. Yap. A path-optimal gac algorithm for table constraints. In *Proceedings of ECAI 2012*, pages 510–515, 2012.

[LLY14] Christophe Lecoutre, Chavalit Likitvivatanavong, and Roland HC Yap. Improving the lower bound of simple tabular reduction. *Constraints*, pages 1–9, 2014.

[LPS13] C. Lecoutre, A. Paparrizou, and K. Stergiou. Extending STR to a higher-order consistency. In *Proceedings of AAAI'13*, pages 576–582, 2013.

[LR05] Olivier Lhomme and Jean-Charles Régin. A fast arc consistency algorithm for n-ary constraints. In *Proceedings of the Nationnal Conference on Artificial Intelligence*, pages 405–410. AAAI Press, 2005.

[LS06] Christophe Lecoutre and Radoslaw Szymanek. Generalized arc consistency for positive table constraints. In *Proceedings of CP 2006*, pages 284–298, 2006.

[LXY14] Chavalit Likitvivatanavong, Wei Xia, and Roland HC Yap. Higher-order consistencies through gac on factor variables. In *Principles and Practice of Constraint Programming*, pages 497–513. Springer, 2014.

[Mac77a] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[Mac77b] Alan K Mackworth. On reading sketch maps. In *IJCAI*, volume 77, pages 598–606, 1977.

[McG79] James J McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19(3):229–250, 1979.

[MDL14] Jean-Baptiste Mairy, Yves Deville, and Christophe Lecoutre. Domain k-wise consistency made as simple as generalized arc consistency. In *Integration of AI and OR Techniques in Constraint Programming*, pages 235–250. Springer International Publishing, 2014.

[MDL15] Jean-Baptiste Mairy, Yves Deville, and Christophe Lecoutre. The smart table constraint. In *Integration of AI and OR Techniques in Constraint Programming*. Springer International Publishing, 2015.

[MDVH11] Jean-Baptiste Mairy, Yves Deville, and Pascal Van Hentenryck. Reinforced adaptive large neighborhood search. *The Seventeenth International Conference on Principles and Practice of Constraint Programming (CP 2011)*, page 55, 2011.

[MF85] Alan K Mackworth and Eugene C Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial intelligence*, 25(1):65–74, 1985.

[MH86] Roger Mohr and Thomas C Henderson. Arc and path consistency revisited. *Artificial intelligence*, 28(2):225–233, 1986.

[MM88] Roger Mohr and Gérald Masini. Good old discrete relaxation. In *Proceedings of ECAI 1988*, pages 651–656, 1988.

[MSD10] Jean-Baptiste Mairy, Pierre Schaus, and Yves Deville. Generic adaptive heuristics for large neighborhood search. In *Seventh International Workshop on Local Search Techniques in Constraint Satisfaction (LSCS2010). A Satellite Workshop of CP*, 2010.

[MT99] P. Meseguer and C. Torras. Solving strategies for highly symmetric CSPs. In *Proceedings of IJCAI'99*, pages 400–405, 1999.

[MVHD12] Jean-Baptiste Mairy, Pascal Van Hentenryck, and Yves Deville. An optimal filtering algorithm for table constraints. In *Principles and Practice of Constraint Programming*, pages 496–511. Springer Berlin Heidelberg, 2012.

[MVHD14a] Jean-Baptiste Mairy, Pascal Van Hentenryck, and Yves Deville. Optimal and efficient filtering algorithms for table constraints. *Constraints*, 19(1):77–120, 2014.

[MVHD14b] Jean-Baptiste Mairy, Pascal Van Hentenryck, and Yves Deville. Optimal and efficient filtering algorithms for table constraints. *Constraints*, 19(1):77–120, 2014.

[NGJM13] Peter Nightingale, Ian Philip Gent, Christopher Anthony Jefferson, and Ian James Miguel. Short and long supports for constraint propagation. *Journal of Artificial Intelligence Research*, 46:1–45, 2013.

[Pes04] Gilles Pesant. A regular language membership constraint for finite sequences of variables. In *Proceedings of CP 2004*, pages 482–495. Springer, 2004.

[PF] Laurent Perron and Vincent Furnon. or-tools. http://code.google.com/p/or-tools.

[PR14] Guillaume Perez and Jean-Charles Régin. Improving GAC-4 for table and MDD constraints. In *Proceedings of CP'14*, pages 606–621, 2014.

[PS12] A. Paparrizou and K. Stergiou. An efficient higher-order consistency algorithm for table constraints. In *Proceedings of AAAI'12*, pages 335–541, 2012.

[QW06] Claude-Guy Quimper and Toby Walsh. Global grammar constraints. In *Proceedings of CP 2006*, pages 751–755. Springer, 2006.

[RBW06] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[Rég94] J. C. Régin. A filtering algorithm for constraints of difference in CSP. In *Proceedings of the AAAI National Conference*, pages 362–367, Seattle, Wash., 1994.

[Rég11] Jean-Charles Régin. Improving the expressiveness of table constraints. In *In proceedings of ModRef 2011 Workshop held with CP 2011*, 2011.

[SO08] H. Simonis and B. O'Sullivan. Search strategies for rectangle packing. In *Proceedings of CP'08*, pages 52–66, 2008.

[SS05] Nikolaos Samaras and Kostas Stergiou. Binary encodings of non-binary constraint satisfaction problems: Algorithms and experimental results. *Journal of Artificial Intelligence Research*, 24(1):641–684, 2005.

[Ste07] K. Stergiou. Strong inverse consistencies for non-binary CSPs. In *Proceedings of ICTAI'07*, pages 215–222, 2007.

[Ste08] K. Stergiou. Strong domain filtering consistencies for non-binary constraint satisfaction problems. *International Journal on Artificial Intelligence Tools*, 17(5):781–802, 2008.

[Ull07] Julian R. Ullmann. Partition search for non-binary constraint satisfaction. *Information Sciences*, 177(18):3639–3678, 2007.

[vDLR] M. van Dongen, C. Lecoutre, and O. Roussel. 2008 CSP solver competition. http://www.cril.univ-artois.fr/CPAI08/.

[VDT92] Pascal Van Hentenryck, Yves Deville, and Choh-Man Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2-3):291–321, 1992.

[VHR95] P. Van Hentenryck and V. Ramachandran. Backtracking without Trailing in CLP($\Re_{lin}$). *ACM Transactions on Programming Languages and Systems*, 17(4):635–671, July 1995.

[VHSD98] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(fd). *The Journal of Logic Programming*, 37(1-3):139–164, 1998.

[VPJ11] J. Vion, T. Petit, and N. Jussien. Integrating strong local consistencies into constraint solvers. *Recent Advances in Constraints*, 6384:90–104, 2011.

[Wal05] Richard Wallace. Factor analytic studies of csp heuristics. In *Proceedings of CP 2005*, volume 3709, pages 712–726. Springer Berlin / Heidelberg, 2005.

[WKCB11] R. Woodward, S. Karakashian, B. Choueiry, and C. Bessière. Solving difficult CSPs with relational neighborhood inverse consistency. In *Proceedings of AAAI'11*, pages 112–119, 2011.

[WM96] Jörg Würtz and Tobias Müller. Constructive disjunction revisited. In *Proceedings of KI'96*, pages 377–386, 1996.

[WPB00] Ron Wehrens, Hein Putter, and Lutgarde Buydens. The bootstrap: a tutorial. *Chemometrics and intelligent laboratory systems*, 54(1):35–52, 2000.

[XBHL07] Ke Xu, Frédéric Boussemart, Fred Hemery, and Christophe Lecoutre. Random constraint satisfaction: Easy generation of hard (satisfiable) instances. *Artificial Intelligence*, 171(8-9):514–534, 2007.

[XY13] Wei Xia and Roland H. C. Yap. Optimizing STR algorithms with tuple compression. In *Proceedings of CP'13*, pages 724–732, 2013.

[You94] G Alastair Young. Bootstrap: More than a stab in the dark? *Statistical Science*, pages 382–395, 1994.

[ZY01] Yuanlin Zhang and Roland H. C. Yap. Making AC3 an Optimal Algorithm. In *International Joint Conference on Artificial Intelligence*, pages 316–321, 2001.